Guided Tutorial GNU Radio in C++

From GNU Radio

Contents

Tutorial: Working with GNU Radio in C++

Objectives

- Extend our knowledge to program GNU Radio using C++.
- An introduction to GNU Radio's C++ API, in particular:
  - types
  - generic functions
  - GNU Radio blocks
- Learn to manage and write our own OOT modules in C++.
  - We follow our discussions based on a working example by continuing to work on our OOT module.

- Within the tutorial module, we will build our QPSK demodulator called as **My QPSK Demodulator** in C++.
- Understand the nuances of developing an OOT module.
  - The tutorial considers some advance topics that are also part of GNU Radio framework.
  - These topics find their usage for some specific implementations of the OOT module.

Prerequisites

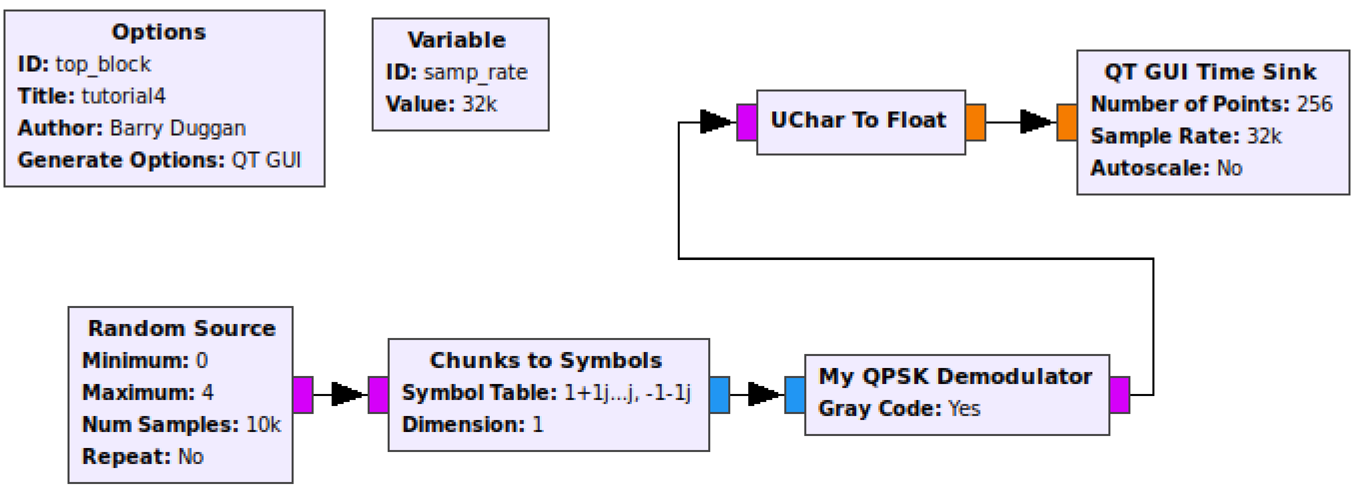
- Knowledge of C++
- Previous tutorials recommended:
  - **A brief introduction to GNU Radio, SDR, and DSP**
  - **Intro to GR usage: GRC and flowgraphs**

Creating our OOT module

We will now use gr_modtool to create an OOT module and write our block in C++.

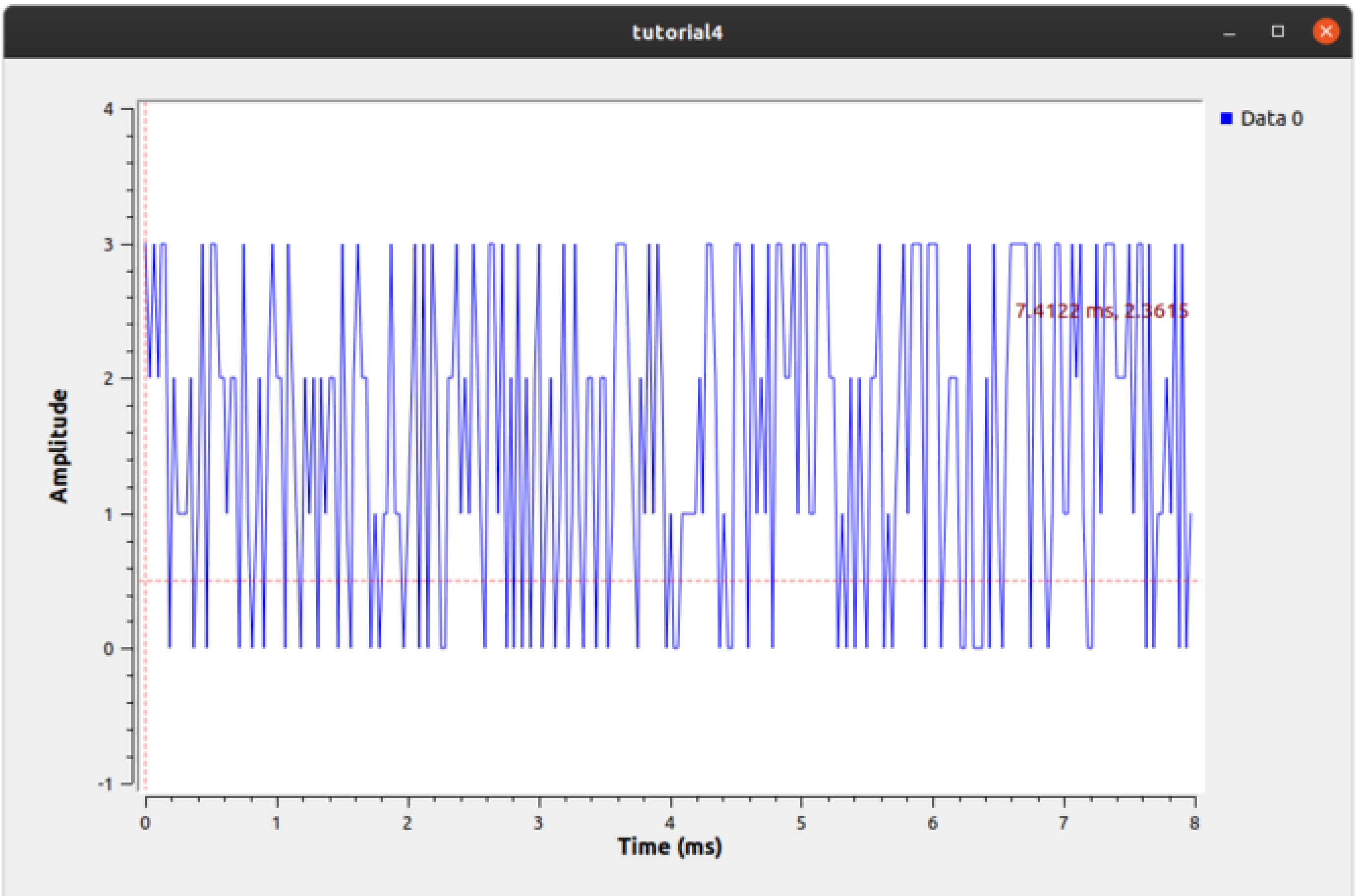
Objective

When this tutorial is complete, we will be able to build this flow graph:



The flowgraph demonstrates a QPSK transceiver chain with the block **My QPSK Demodulator** block module under the OOT **tutorial**. We will be building this block using C++. All other blocks are standard GNU Radio blocks.

My QPSK Demodulator consumes QPSK symbols which are complex values at the input and produces the alphabets as bytes at output. The output screen looks like this:



Step 1: Create an OOT module `gr-tutorial`

```
xyz@comp:mydir$ gr_modtool newmod tutorial
Creating out-of-tree module in ./gr-tutorial... Done.
Use 'gr_modtool add' to add a new block to this currently empty module.
```

Take a look at the directory structure of our `gr-tutorial`

```
xyz@comp:mydir$ cd ~/gr-tutorial
xyz@comp:mydir/gr-tutorial$ ls
apps  cmake  CMakeLists.txt  docs  examples  grc  include  lib  MANIFEST.md  python  swig
```

Note: The file MANIFEST.md was not added prior to GR 3.8.

Step 2: Insert My QPSK Demodulator block into the OOT module

Again using `gr_modtool`, inside `gr-tutorial`, we create our `my_qpsk_demod` block:

```
xyz@comp:mydir/gr-tutorial$ gr_modtool add my_qpsk_demod_cb
GNU Radio module name identified: tutorial
('sink', 'source', 'sync', 'decimator', 'interpolator', 'general', 'tagged_stream', 'hier', 'noblock')
Enter block type: general
Language (python/cpp): cpp
Language: C++
Block/code identifier: my_qpsk_demod_cb
Please specify the copyright holder: gnuradio.org
```

```
Enter valid argument list, including default arguments:
bool gray_code
Add Python QA code? [Y/n] Y
Add C++ QA code? [y/N] N
Adding file 'lib/my_qpsk_demod_cb_impl.h'...
Adding file 'lib/my_qpsk_demod_cb_impl.cc'...
Adding file 'include/tutorial/my_qpsk_demod_cb.h'...
Editing swig/tutorial_swig.i...
Adding file 'python/qa_my_qpsk_demod_cb.py'...
Editing python/CMakeLists.txt...
Adding file 'grc/tutorial_my_qpsk_demod_cb.block.yml'...
Editing grc/CMakeLists.txt...
```

Unlike creating an OOT module, creating a block using `gr_modtool` demands inputs from the user. To follow the command line user interaction, let's decompose the information above.

```
xyz@comp:mydir/gr-tutorial$ gr_modtool add my_qpsk_demod_cb
```

`my_qpsk_demod_cb` represents the class name of the block, where the suffix, 'cb' is added to the block name, which conforms to the GNU Radio nomenclature. 'cb' states the block takes complex data as input and produces bytes as output.

```
Enter code type: general
```

In GNU Radio, there exist different kinds of blocks with the different possibilities listed above (since 3.8). Depending on the choice of our block, `gr_modtool` adds the corresponding code and functions. As illustrated, for `my_qpsk_demod_cb` block, we opt for a general block.

```
Please specify the copyright holder:
```

Since version 3.8, the tool asks for a copyright holder for the code you are about to write. The implications of what you write in that line are legal and not computational.

For `my_qpsk_demod_cb`, gray_code is selected to be "default arguments".

```
Enter valid argument list, including default arguments:  bool gray_code
```

GNU Radio provides an option of writing test cases. This provides quality assurance to the code written. If selected, the `gr_modtool` adds the quality assurance files corresponding to python and C++.

```
Add Python QA code? [Y/n]
Add C++ QA code? [y/N] y
```

With this, we have already established the GNU Radio semantics for our block coupled with the OOT module. In the following sections, we will focus on the implementation of our block.

The detailed description of coding structure for the block can be found here.

Step 3: Fleshing out the code

The next step is to implement the logic for our block. Use your text editor on the file `~/gr-tutorial/lib/my_qpsk_demod_cb_impl.cc`.

The completed code is listed at the end of this section, but study the following first!

The skeleton of the `my_qpsk_demod_cb_impl.cc` has the following structure:

```
  /*!
   * The private constructor
   */
  my_qpsk_demod_cb_impl::my_qpsk_demod_cb_impl(bool gray_code)
    : gr::block("my_qpsk_demod_cb",
            gr::io_signature::make(<+MIN_IN+>, <+MAX_IN+>, sizeof(<+ITYPE+>)),
            gr::io_signature::make(<+MIN_OUT+>, <+MAX_OUT+>, sizeof(<+OTYPE+>)))
  {}
```

- `my_qpsk_demod_cb_impl()` is the constructor of the block `my_qpsk_demod`. `my_qpsk_demod_cb_impl()` calls the constructor of the base class block `gr::block(...)` defined here (http://gnuradio.org/doc/doxygen/basic__block_8h_source.html).
- The arguments inside `gr::block(...)` represents the block name and a call to the make function.
- The make function `gr::io_signature::make(<+MIN_IN+>, <+MAX_IN+>, sizeof(<+ITYPE+>))` and `gr::io_signature::make(<+MIN_OUT+>, <+MAX_OUT+>, sizeof(<+OTYPE+>))` is a member function of the class `gr::io_signature` that signifies the input and output port/s.
- *<MIN_OUT>* and *<MAX_OUT>* represents the maximum and number of ports.
- <ITYPE> and <OTYPE> indicates the datatypes for the input and output port/s which needs to be filled out manually.

Next, we need to modify the constructor. After modification, it looks like this:

```
  /*!
   * The private constructor
   */
  my_qpsk_demod_cb_impl::my_qpsk_demod_cb_impl(bool gray_code)
    : gr::block("my_qpsk_demod_cb",
            gr::io_signature::make(1, 1, sizeof(gr_complex)),
            gr::io_signature::make(1, 1, sizeof(char))),
      d_gray_code(gray_code)
  {}
```

The option `gray_code` is copied to the class attribute `d_gray_code`. Note that we need to declare this a private member of the class in the header file `my_qpsk_demod_cb_impl.h`,

```
  private:
    bool d_gray_code;
```

Also inside this class is the method `general_work()`, which is pure virtual in `gr::block`, so we definitely need to override that. After running `gr_modtool`, the skeleton version of this function will look something like this:

```
int
my_qpsk_demod_cb_impl::general_work (int noutput_items,
                   gr_vector_int &ninput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
```

```
{
    const <+ITYPE*> *in = (const <+ITYPE*> *) input_items[0];
    <+OTYPE*> *out = (<+OTYPE*> *) output_items[0];

    // Do <+signal processing+>
    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

There is one pointer to the input- and one pointer to the output buffer, respectively, and a for-loop which processes the items in the input buffer and copies them to the output buffer. Once the demodulation logic is implemented, the structure of the work function has the form

```
int
my_qpsk_demod_cb_impl::general_work (int noutput_items,
                   gr_vector_int &ninput_items,
                   gr_vector_const_void_star &input_items,
                   gr_vector_void_star &output_items)
{
    const gr_complex *in = (const gr_complex *) input_items[0];
    unsigned char *out = (unsigned char *) output_items[0];
    gr_complex origin = gr_complex(0,0);
    // Perform ML decoding over the input iq data to generate alphabets
    for(int i = 0; i < noutput_items; i++)
    {
            // ML decoder, determine the minimum distance from all constellation points
            out[i] = get_minimum_distances(in[i]);
    }

    // Tell runtime system how many input items we consumed on
    // each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

This work function calls another function `get_minimum_distances(const gr_complex &sample)`, which we also need to add:

```
unsigned char
my_qpsk_demod_cb_impl::get_minimum_distances(const gr_complex &sample)
{
  if (d_gray_code) {
    unsigned char bit0 = 0;
    unsigned char bit1 = 0;
    // The two left quadrants (quadrature component < 0) have this bit set to 1
    if (sample.real() < 0) {
      bit0 = 0x01;
    }
    // The two lower quadrants (in-phase component < 0) have this bit set to 1
    if (sample.imag() < 0) {
      bit1 = 0x01 << 1;
    }
    return bit0 | bit1;
  } else {
    // For non-gray code, we can't simply decide on signs, so we check every single quadrant.
    if (sample.imag() >= 0 and sample.real() >= 0) {
      return 0x00;
    }
    else if (sample.imag() >= 0 and sample.real() < 0) {
      return 0x01;
    }
    else if (sample.imag() < 0 and sample.real() < 0) {
```

```
          return 0x02;
        }
        else if (sample.imag() < 0 and sample.real() >= 0) {
          return 0x03;
        }
      }
    }
```

Note: the get_minimum_distances function declaration also needs to be added to the class header (`my_qpsk_demod_cb_impl.h`).

The function `get_minimum_distances` is a maximum likelihood decoder for the QPSK demodulater. Theoretically, the function should compute the distance from each ideal QPSK symbol to the received symbol (It is mathematically equivalent to determining the Voronoi regions of the received sample). For a QPSK signal, these Voronoi regions are simply four quadrants in the complex plane. Hence, to decode the sample into bits, it makes sense to map the received sample to these quadrants.

Now, let's consider the `forecast()` function. The system needs to know how much data is required to ensure validity in each of the input arrays. As stated before, the `forecast()` method provides this information, and you must therefore override it anytime you write a `gr::block` derivative (for sync blocks, this is implicit).

The default implementation of `forecast()` says there is a 1:1 relationship between noutput_items and the requirements for each input stream. The size of the items is defined by `gr::io_signature::make` in the constructor of `gr::block`. The sizes of the input and output items can of course differ; this still qualifies as a 1:1 relationship. Of course, if you had this relationship, you wouldn't want to use a `gr::block`!

```
// default implementation:  1:1
void
gr::block::forecast(int noutput_items,
                    gr_vector_int &ninput_items_required)
{
  unsigned ninputs = ninput_items_required.size ();
  for(unsigned i = 0; i < ninputs; i++)
    ninput_items_required[i] = noutput_items;
}
```

Although the 1:1 implementation worked for `my_qpsk_demod_cb`, it wouldn't be appropriate for interpolators, decimators, or blocks with a more complicated relationship between `noutput_items` and the input requirements. That said, by deriving your classes from `gr::sync_block`, `gr::sync_interpolator` or `gr::sync_decimator` instead of `gr::block`, you can often avoid implementing `forecast`.

Refilling the private constructor and overriding the `general_work()` and `forecast()` will suffice the coding structure of our block. However, in the `gr::block` class there exists more specific functions. These functions are covered under advanced topics section (http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorial_GNU_Radio_in_C++#Advanced-topics)

Here is the completed source code:

```
/* -*- c++ -*- */
/*
 * my_qpsk_demod_cb_impl.h
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
```

```
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifndef INCLUDED_TUTORIAL_MY_QPSK_DEMOD_CB_IMPL_H
#define INCLUDED_TUTORIAL_MY_QPSK_DEMOD_CB_IMPL_H

#include <tutorial/my_qpsk_demod_cb.h>

namespace gr {
  namespace tutorial {

    class my_qpsk_demod_cb_impl : public my_qpsk_demod_cb
    {
    private:
      bool d_gray_code;

     public:
      my_qpsk_demod_cb_impl(bool gray_code);
      ~my_qpsk_demod_cb_impl();
      unsigned char get_minimum_distances(const gr_complex &sample);

      // Where all the action really happens
      void forecast (int noutput_items, gr_vector_int &ninput_items_required);

      int general_work(int noutput_items,
           gr_vector_int &ninput_items,
           gr_vector_const_void_star &input_items,
           gr_vector_void_star &output_items);
    };

  } // namespace tutorial
} // namespace gr

#endif /* INCLUDED_TUTORIAL_MY_QPSK_DEMOD_CB_IMPL_H */
```

```
/* -*- c++ -*- */
/*
 * my_qpsk_demod_cb_impl.cc
 *
 * This is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * This software is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

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

#include <gnuradio/io_signature.h>
#include "my_qpsk_demod_cb_impl.h"

namespace gr {
  namespace tutorial {

```

```
    my_qpsk_demod_cb::sptr
    my_qpsk_demod_cb::make(bool gray_code)
    {
      return gnuradio::get_initial_sptr
        (new my_qpsk_demod_cb_impl(gray_code));
    }

    /*
     * The private constructor
     */
    my_qpsk_demod_cb_impl::my_qpsk_demod_cb_impl(bool gray_code)
      : gr::block("my_qpsk_demod_cb",
              gr::io_signature::make(1, 1, sizeof(gr_complex)),
              gr::io_signature::make(1, 1, sizeof(char))),
        d_gray_code(gray_code)
    {
    }

    /*
     * Our virtual destructor.
     */
    my_qpsk_demod_cb_impl::~my_qpsk_demod_cb_impl()
    {
    }

    void
    my_qpsk_demod_cb_impl::forecast(int noutput_items,
                     gr_vector_int &ninput_items_required)
    {
    unsigned ninputs = ninput_items_required.size ();
    for(unsigned i = 0; i < ninputs; i++)
      ninput_items_required[i] = noutput_items;
    }

    unsigned char
    my_qpsk_demod_cb_impl::get_minimum_distances(const gr_complex &sample)
    {
      if (d_gray_code) {
        unsigned char bit0 = 0;
        unsigned char bit1 = 0;
        // The two left quadrants (quadrature component < 0) have this bit set to 1
        if (sample.real() < 0) {
          bit0 = 0x01;
        }
        // The two lower quadrants (in-phase component < 0) have this bit set to 1
        if (sample.imag() < 0) {
          bit1 = 0x01 << 1;
        }
        return bit0 | bit1;
      } else {
        // For non-gray code, we can't simply decide on signs, so we check every single quadrant.
        if (sample.imag() >= 0 and sample.real() >= 0) {
          return 0x00;
        }
        else if (sample.imag() >= 0 and sample.real() < 0) {
          return 0x01;
        }
        else if (sample.imag() < 0 and sample.real() < 0) {
          return 0x02;
        }
        else if (sample.imag() < 0 and sample.real() >= 0) {
          return 0x03;
        }
      }
    }

    int
    my_qpsk_demod_cb_impl::general_work (int noutput_items,
                       gr_vector_int &ninput_items,
                       gr_vector_const_void_star &input_items,
                       gr_vector_void_star &output_items)
    {
        const gr_complex *in = (const gr_complex *) input_items[0];
        unsigned char *out = (unsigned char *) output_items[0];
        gr_complex origin = gr_complex(0,0);
```

```
        // Perform ML decoding over the input iq data to generate alphabets
        for(int i = 0; i < noutput_items; i++)
        {
                // ML decoder, determine the minimum distance from all constellation points
                out[i] = get_minimum_distances(in[i]);
        }
        // Tell runtime system how many input items we consumed on
        // each input stream.
        consume_each (noutput_items);
        // Tell runtime system how many output items we produced.
        return noutput_items;
    }

  } /* namespace tutorial */
} /* namespace gr */
```

Step 4: Flesh out the XML file

In GNU Radio 3.7 the .xml provides the user interface between the OOT module displayed in the GRC and the source code. For GNU Radio 3.8 and newer, see the next section which describes the YAML format. Moreover, the XML file defines an interface to pass the parameters specific for the module. Hence, to access the module inside GRC, it is important to modify the .xml files manually. The XML file for our block is named as `demod_my_qpsk_demod_cb.xml` inside the `grc/` folder. Presently, the `gr_modtool`'s version looks like:

Default version:

```
<?xml version="1.0"?>
<block>
  <name>my_qpsk_demod_cb</name>
  <key>tutorial_my_qpsk_demod_cb</key>
  <category>tutorial</category>
  <import>import tutorial</import>
  <make>tutorial.my_qpsk_demod_cb($gray_code)</make>
  <!-- Make one 'param' node for every Parameter you want settable from the GUI.
       Sub-nodes:
       * name
       * key (makes the value accessible as $keyname, e.g. in the make node)
       * type -->
  <param>
    <name>...</name>
    <key>...</key>
    <type>...</type>
  </param>

  <!-- Make one 'sink' node per input. Sub-nodes:
       * name (an identifier for the GUI)
       * type
       * vlen
       * optional (set to 1 for optional inputs) -->
  <sink>
    <name>in</name>
    <type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
  </sink>

  <!-- Make one 'source' node per output. Sub-nodes:
       * name (an identifier for the GUI)
       * type
       * vlen
       * optional (set to 1 for optional inputs) -->
  <source>
    <name>out</name>
    <type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
  </source>
</block>
```

The parameter `gray_code` can be put under the `<parameter>` tag.

Adding parameter tag:

```
<?xml version="1.0"?>
<param>
  <name>Gray Code</name>
  <key>gray_code</key>
  <value>True</value>
  <type>bool</type>
  <option>
    <name>Yes</name>
    <key>True</key>
  </option>
  <option>
    <name>No</name>
    <key>False</key>
  </option>
</param>
```

Like the work function, the datatypes for the input and output ports represented by `<sink>` and `<source>` tags should be modified.

Modifying source and sink tag:

```
<sink>
   <name>in</name>
   <type>complex</type>
 </sink>
```

```
<source>
    <name>out</name>
    <type>byte</type>
  </source>
```

After all the necessary modification the "tutorial_my_qpsk_demod_cb.xml" looks like this:

Modified version:

```
<?xml version="1.0"?>
<block>
  <name>My QPSK Demodulator</name>
  <key>tutorial_my_qpsk_demod_cb</key>
  <category>tutorial</category>
  <import>import tutorial</import>
  <make>tutorial.my_qpsk_demod_cb($gray_code)</make>
  <param>
    <name>Gray Code</name>
    <key>gray_code</key>
    <value>True</value>
    <type>bool</type>
    <option>
      <name>Yes</name>
      <key>True</key>
```

```
    </option>
    <option>
      <name>No</name>
      <key>False</key>
    </option>
  </param>
  <sink>
    <name>in</name>
    <type>complex</type>
  </sink>
  <source>
    <name>out</name>
    <type>byte</type>
  </source>
</block>
```

Step 4 bis: Flesh out the YAML file

Since version 3.8, GNU Radio has replaced the XML files by YAML. They work the same but with a different syntax.

The .yml provides the user interface between the OOT module displayed in the GRC and the source code. Moreover, the YAML file defines an interface to pass the parameters specific for the module. Hence, to access the module inside GRC, it is important to modify the .yml files manually. The YAML file for our block is named as `tutorial_my_qpsk_demod_cb.block.yml` inside the `grc/` folder. Presently, the `gr_modtool`'s version looks like:

Default version:

```
id: tutorial_my_qpsk_demod_cb
label: my_qpsk_demod_cb
category: '[tutorial]'

templates:
  imports: import tutorial
  make: tutorial.my_qpsk_demod_cb(${gray_code})

#  Make one 'parameters' list entry for every parameter you want settable from the GUI.
#     Keys include:
#     * id (makes the value accessible as \$keyname, e.g. in the make entry)
#     * label (label shown in the GUI)
#     * dtype (e.g. int, float, complex, byte, short, xxx_vector, ...)
parameters:
- id: ...
  label: ...
  dtype: ...
- id: ...
  label: ...
  dtype: ...

#  Make one 'inputs' list entry per input and one 'outputs' list entry per output.
#  Keys include:
#      * label (an identifier for the GUI)
#      * domain (optional - stream or message. Default is stream)
#      * dtype (e.g. int, float, complex, byte, short, xxx_vector, ...)
#      * vlen (optional - data stream vector length. Default is 1)
#      * optional (optional - set to 1 for optional inputs. Default is 0)
inputs:
- label: ...
  domain: ...
  dtype: ...
  vlen: ...
  optional: ...

outputs:
- label: ...
  domain: ...
```

```
  dtype: ...
  vlen: ...
  optional: ...

#  'file_format' specifies the version of the GRC yml format used in the file
#  and should usually not be changed.
file_format: 1
```

The parameter `gray_code` can be put under the `parameters` tag.

Adding parameter tag:

```
parameters:
- id: gray_code
  label: Gray Code
  dtype: bool
  default: 'True'
```

Like the work function, the datatypes for the input and output ports represented by `input` and `output` tags should be modified.

Modifying source and sink tag:

```
inputs:
- label: in
  dtype: complex
```

```
outputs:
- label: out
  dtype: byte
```

After all the necessary modification the "tutorial_my_qpsk_demod_cb.block.yml" looks like this:

Modified version:

```
id: tutorial_my_qpsk_demod_cb
label: My QPSK Demodulator
category: '[tutorial]'

templates:
  imports: import tutorial
  make: tutorial.my_qpsk_demod_cb(${gray_code})

parameters:
- id: gray_code
  label: Gray Code
  dtype: bool
  default: 'True'

inputs:
- label: in
  dtype: complex
outputs:
- label: out
  dtype: byte
```

```
file_format: 1
```

Step 5: Install my_qpsk_demod in grc

Now that we have finished the implementation of our block, we need to build and install it. To do so, execute the following commands:

```
cd ~/gr-tutorial
mkdir build
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

Step 6: Quality Assurance (Unit Testing)

In the previous steps of writing the OOT module, we produced the QPSK demodulator, but it doesn't guarantee the correct working of our block. In this situation, it becomes significant to write a unit test for our module that certifies the clean implementation of the QPSK demodulator.

Below is the source of code of the `qa_qpsk_demod.py` can be found under `python/`

Full QA code

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
import tutorial_swig as tutorial
from numpy import array

class qa_qpsk_demod (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_gray_code_enabled (self):
        # "Construct the Iphase and Qphase components"
        Iphase = array([ 1, -1, -1,  1])
        Qphase = array([ 1,  1, -1, -1])
        src_data = Iphase + 1j*Qphase;
        # "Enable Gray code"
        gray_code =  True;
        # "Determine the expected result"
        expected_result = (0,1,3,2)
        # "Create a complex vector source"
        src = blocks.vector_source_c(src_data)
        # "Instantiate the test module"
        qpsk_demod = tutorial.my_qpsk_demod_cb(gray_code)
        # "Instantiate the binary sink"
        dst = blocks.vector_sink_b();
        # "Construct the flowgraph"
        self.tb.connect(src,qpsk_demod)
        self.tb.connect(qpsk_demod,dst)
        # "Create the flow graph"
        self.tb.run ()
        # check data
        result_data = dst.data()
        self.assertTupleEqual(expected_result, result_data)
        self.assertEqual(len(expected_result), len(result_data))
```

```

    def test_002_gray_code_disabled (self):
        # "Construct the Iphase and Qphase components"
        Iphase = array([ 1, -1, -1,  1])
        Qphase = array([ 1,  1, -1, -1])
        src_data = Iphase + 1j*Qphase;
        # "Enable Gray code"
        gray_code =  False;
        # "Determine the expected result"
        expected_result = (0,1,2,3)
        # "Create a complex vector source"
        src = blocks.vector_source_c(src_data)
        # "Instantiate the test module"
        qpsk_demod = tutorial.my_qpsk_demod_cb(gray_code)
        # "Instantiate the binary sink"
        dst = blocks.vector_sink_b();
        # "Construct the flowgraph"
        self.tb.connect(src,qpsk_demod)
        self.tb.connect(qpsk_demod,dst)
        # "Create the flow graph"
        self.tb.run ()
        # check data
        result_data = dst.data()
        self.assertTupleEqual(expected_result, result_data)
        self.assertEqual(len(expected_result), len(result_data))

if __name__ == '__main__':
    gr_unittest.run(qa_qpsk_demod, "qa_qpsk_demod.xml")
```

Obviously the `qa_qpsk_demod` is implemented in python, in spite of we opted C++ in the first case for writing our blocks. This is because GNU Radio inherits the python unittest framework (https://docs.python.org/2/library/unittest.html) to support quality assurance. And, if you remember it correctly from previous tutorials, swig as part of GNU Radio framework, provides python bindings for the C++ code. Hence, we are able to write the unit test for our block `qa_qpsk_demod` in Python.

So lets gather a bit of know how on how to write test cases for the block. Okay, lets consider the header part first:

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
import tutorial_swig as tutorial
from numpy import array
```

`from gnuradio import gr, gr_unittest` and `from gnuradio import blocks` are the standard lines that includes gr, gr_unittest functionality in the `qa_` file. `import tutorial_swig as tutorial` import the python bidden version of our module, which provides an access our block `my_qpsk_demod_cb`. Finally, `from numpy import array` includes array.

```
if __name__ == '__main__':
    gr_unittest.run(qa_qpsk_demod, "qa_qpsk_demod.xml")
```

The `qa_` file execution start by calling this function. The `gr_unittest` automatically calls the functions in a specific order `def setUp (self)` for creating the top block at the start, `tearDown (self)` for deleting the top block at the end. In between the `setUp` and `tearDown` the test cases defined are executed. The methods starting with prefix `test_` are recognized as test cases by `gr_unittest`. We have defined two test cases `test_001_gray_code_enabled` and `test_002_gray_code_disabled`. The usual structure of a test cases comprises of a known input data and the expected output. A flowgraph is created to include the source (input data), block to be tested (processor) and sink (resulted output data). In the end the expected output is compared with the resulted output data.

Finally, the statements in the test cases

```
self.assertTupleEqual(expected_result, result_data)
self.assertEqual(len(expected_result), len(result_data))
```

determine the result of test cases as passed or failed. The test cases are executed before installation of the module by running `make test` yielding the following output:

```
barry@barry:~/gr-tutorial/build$ make test
Running tests...
Test project /home/barry/gr-tutorial/build
    Start 1: test_tutorial
1/2 Test #1: test_tutorial ....................   Passed    0.02 sec
    Start 2: qa_my_qpsk_demod_cb
2/2 Test #2: qa_my_qpsk_demod_cb .............   Passed    1.01 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   1.04 sec
```

Congratulations, we have just finished writing our OOT module `gr-tutorial` and a C++ block `my_qpsk_demodulator`.

Advanced topics

The topics discussed until now have laid the foundation for designing the OOT module independently. However, the GNU Radio jargon extends further beyond these. Therefore, under this section, we drift from the QPSK demodulator and focus on the features that are rarely used or are more specific to the implementation.

To add physical meaning to the discussion, we have taken assistance of the existing modules. The source code excerpts are included thereof. Enthusiastic readers are suggested to open the source code in parallel and play around with their functionalities.

Specific functions related to block

In the last section, we managed out implementation of our block by defining functions like `general_work` and `forecast()`. But sometimes special functions need to be defined for the implementation. The list is long, but we try to discuss same of these functions in the following subsections.

set_history()

If your block needs a history (i.e., something like an FIR filter), call this in the constructor.
Here is an example

```
test::test(const std::string &name,
            int min_inputs, int max_inputs,
            unsigned int sizeof_input_item,
            int min_outputs, int max_outputs,
            unsigned int sizeof_output_item,
            unsigned int history,
            unsigned int output_multiple,
            double relative_rate,
            bool fixed_rate,
            consume_type_t cons_type, produce_type_t prod_type)
```

```
  : block (name,
          io_signature::make(min_inputs, max_inputs, sizeof_input_item),
          io_signature::make(min_outputs, max_outputs, sizeof_output_item)),
    d_sizeof_input_item(sizeof_input_item),
    d_sizeof_output_item(sizeof_output_item),
    d_check_topology(true),
    d_consume_type(cons_type),
    d_min_consume(0),
    d_max_consume(0),
    d_produce_type(prod_type),
    d_min_produce(0),
    d_max_produce(0)
  {
    set_history(history);
    set_output_multiple(output_multiple);
    set_relative_rate(relative_rate);
    set_fixed_rate(fixed_rate);
  }
```

GNU Radio then makes sure you have the given number of 'old' items available.

The smallest history you can have is 1, i.e., for every output item, you need 1 input item. If you choose a larger value, N, this means your output item is calculated from the current input item and from the N-1 previous input items.

The scheduler takes care of this for you. If you set the history to length N, the first N items in the input buffer include the N-1 previous ones (even though you've already consumed them).

The history is stored in the variable `d_history`.
The `set_history()` is defined in `gnuradio/gnuradio-runtime/block.cc`

```
void block::set_history(unsigned history)
{
    d_history = history;
}
```

set_output_multiple()

When implementing your `general_work()` routine, it's occasionally convenient to have the run time system ensure that you are only asked to produce a number of output items that is a multiple of some particular value. This might occur if your algorithm naturally applies to a fixed sized block of data. Call `set_output_multiple` in your constructor to specify this requirement,

{{collapse(code)

```
test::test(const std::string &name,
             int min_inputs, int max_inputs,
             unsigned int sizeof_input_item,
             int min_outputs, int max_outputs,
             unsigned int sizeof_output_item,
             unsigned int history,
             unsigned int output_multiple,
             double relative_rate,
             bool fixed_rate,
             consume_type_t cons_type, produce_type_t prod_type)
  : block (name,
          io_signature::make(min_inputs, max_inputs, sizeof_input_item),
          io_signature::make(min_outputs, max_outputs, sizeof_output_item)),
    d_sizeof_input_item(sizeof_input_item),
    d_sizeof_output_item(sizeof_output_item),
```

```
    d_check_topology(true),
    d_consume_type(cons_type),
    d_min_consume(0),
    d_max_consume(0),
    d_produce_type(prod_type),
    d_min_produce(0),
    d_max_produce(0)
{
    set_history(history);
    set_output_multiple(output_multiple);
    set_relative_rate(relative_rate);
    set_fixed_rate(fixed_rate);
}
```

}}

by invoking `set_output_multiple`, we set the value variable to `d_output_multiple`. The default value of `d_output_multiple` is 1.

Lets consider an example, say we want to generate outputs only in a 64 elements chunk, by setting d_output_multiple to 64 we can achieve this, but note that we can also get multiples of 64 i.e. 128, 256 etc

The definition of `set_output_multiple` can be found in gnuradio/gnuradio-runtime/block.cc

```
void gr_block::set_output_multiple (int multiple)
{
  if (multiple < 1)
    throw std::invalid_argument ("gr_block::set_output_multiple");

  d_output_multiple_set = true;
  d_output_multiple = multiple;
}
```

Specific block categories

Again the implementation of the `my_qpsk_demod_cb` was done using a general block. However, GNU Radio includes some blocks with special functionality. A brief overview of these blocks is described in the table.

Block	Functionality
General	This block a generic version of all blocks
Source/Sinks	The source/sink produce/consume the input/output items
Interpolation/Decimation	The interpolation/decimation block is another type of fixed rate block where the number of output/input items is a fixed multiple of the number of input/output items.
Sync	The sync block allows users to write blocks that consume and produce an equal number of items per port. From the user perspective, the GNU Radio scheduler synchronizes the input and output items, it has nothing to with synchronization algorithms
Hierarchical blocks	Hierarchical blocks are blocks that are made up of other blocks.

In the next subsections we discuss these blocks in detail. Again, enthusiastic readers can find these blocks in the GNU Radio source code.

General

```
howto_square_ff::howto_square_ff ()
: gr::block("square_ff",
```

```
gr::io_signature::make(MIN_IN, MAX_IN, sizeof (float)),
gr::io_signature::make(MIN_OUT, MAX_OUT, sizeof (float)))
{
// nothing else required in this example
}
```

Source and Sinks

Source

An example of source block in C++

```
usrp_source_impl::usrp_source_impl(const ::uhd::device_addr_t &device_addr,
                                   const ::uhd::stream_args_t &stream_args):
      sync_block("gr uhd usrp source",
                    io_signature::make(0, 0, 0),
                    args_to_io_sig(stream_args)),
      _stream_args(stream_args),
      _nchan(stream_args.channels.size()),
      _stream_now(_nchan == 1),
      _tag_now(false),
      _start_time_set(false)
```

Some observations:

- `io_signature::make(0, 0, 0)` sets the input items to 0, in indicates there are no input streams.
- Because it connected with the hardware USRP, the `gr uhd usrp source` is a sub class of `sync_block`.

Sink

An example of the sink block in C++

```
usrp_sink_impl::usrp_sink_impl(const ::uhd::device_addr_t &device_addr,
                                   const ::uhd::stream_args_t &stream_args)
      : sync_block("gr uhd usrp sink",
                      args_to_io_sig(stream_args),
                      io_signature::make(0, 0, 0)),
        _stream_args(stream_args),
        _nchan(stream_args.channels.size()),
        _stream_now(_nchan == 1),
        _start_time_set(false)
```

Some observations:

- `io_signature::make(0, 0, 0)` sets the output items to 0, in indicates there are no output streams.
- Because it connected with the hardware USRP, the `gr uhd usrp sink` is a sub class of `sync_block`.

Sync

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 <gnuradio/sync_block.h>

class my_sync_block : public gr_sync_block
{
public:
  my_sync_block(...):
    gr_sync_block("my block",
                  gr::io_signature::make(1, 1, sizeof(int32_t)),
                  gr::io_signature::make(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::io_signature::make(0, 0, 0) makes this a source block
- an output signature of gr::io_signature::make(0, 0, 0) makes this a sink block

Rate changing blocks: Interpolation and Decimation

Decimation

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 must assume that the number of input items = noutput_items*decimation. The value `ninput_items` is therefore implicit.

Interpolation

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 <gnuradio/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...
};
```

Some observations:

- The gr_sync_interpolator constructor takes a 4th parameter, the interpolation factor
- The user must assume that the number of input items = noutput_items/interpolation

Hierarchical blocks

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.

When to use hierarchical blocks?

Hierarchical blocks provides us modularity in our flowgraphs by abstracting simple blocks, that is hierarchical block helps us define our specific blocks at the same time provide us the flexibility to change it, example, we would like to test effect of different modulation schemes for a given channel model. However our synchronization algorithms are specific or newly published. We define our hier block as gr-my_sync that does synchronization followed equalizer and demodulation. We start with BPSK, the flowgraph looks like

gr-tx ---> gr-channel --> gr-my_sync --> gr-equalizer --> gr-bpsk_demod

Now, our flowgraph looks decent. Secondly, we abstracted the complex functionality of our synchronization. Shifting to QPSK, where the synchronization algorithm remains the same, we just replace the gr-bpsk_demod with gr-qpsk_demod

gr-tx ---> gr-channel --> gr-my_sync --> gr-equalizer --> gr-qpsk_demod

How to build hierarchical blocks in GNU Radio?

Hierarchical blocks define an input and output stream much like normal blocks. For **I** input streams, let **i** be a value between 0 and **I**-1. 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>))
```

An typical example of a hierarchical block is OFDM Receiver implemented in python under

```
gnuradio/gr-digital/python/digital
```

The class is defined as:

```
class ofdm_receiver(gr.hier_block2)
```

and instantiated as

```
gr.hier_block2.__init__(self, "ofdm_receiver",
                        gr.io_signature(1, 1, gr.sizeof_gr_complex), # Input signature
                        gr.io_signature2(2, 2, gr.sizeof_gr_complex*occupied_tones, gr.sizeof_char)) # Output signature
```

Some main tasks performed by the OFDM receiver include channel filtering, synchronization and IFFT tasks. The individual tasks are defined inside the hierarchical block.

- Channel filtering

```
chan_coeffs = filter.firdes.low_pass (1.0,                     # gain
                                      1.0,                     # sampling rate
                                      bw+tb,                   # midpoint of trans. band
                                      tb,                      # width of trans. band
                                      filter.firdes.WIN_HAMMING)   # filter type
self.chan_filt = filter.fft_filter_ccc(1, chan_coeffs)
```

- Synchronization

```
self.chan_filt = blocks.multiply_const_cc(1.0)
nsymbols = 18      # enter the number of symbols per packet
freq_offset = 0.0  # if you use a frequency offset, enter it here
nco_sensitivity = -2.0/fft_length    # correct for fine frequency
self.ofdm_sync = ofdm_sync_fixed(fft_length,
                                 cp_length,
                                 nsymbols,
                                 freq_offset,
                                 logging)
```

- ODFM demodulation

```
self.fft_demod = gr_fft.fft_vcc(fft_length, True, win, True)
```

Finally, the individual blocks along with hierarchical are connected among each to form a flow graph.

Connection between the hierarchical block OFDM receiver to channel filter block

```
self.connect(self, self.chan_filt) # filter the input channel
```

Connection between the channel filter block to the OFDM synchronization block.

```
self.connect(self.chan_filt, self.ofdm_sync)
```

and so forth.

Hierarchical blocks can also be nested, that is blocks defined in hierarchical blocks could also be hierarchical blocks. For example, OFDM sync block is also an hierarchical block. In this particular case it is implemented in `C++`. Lets have a look into it.

Underneath is instant of the hierarchical block. Don't panic by looking at its size, we just need to grab the concept behind creating hierarchical blocks.

OFDM impl

```
ofdm_sync_sc_cfb_impl::ofdm_sync_sc_cfb_impl(int fft_len, int cp_len, bool use_even_carriers)
        : hier_block2 ("ofdm_sync_sc_cfb",
                       io_signature::make(1, 1, sizeof (gr_complex)),
#ifndef SYNC_ADD_DEBUG_OUTPUT
                   io_signature::make2(2, 2, sizeof (float), sizeof (unsigned char)))
#else
                   io_signature::make3(3, 3, sizeof (float), sizeof (unsigned char), sizeof (float)))
#endif
    {
      std::vector ma_taps(fft_len/2, 1.0);
      gr::blocks::delay::sptr          delay(gr::blocks::delay::make(sizeof(gr_complex), fft_len/2));
      gr::blocks::conjugate_cc::sptr   delay_conjugate(gr::blocks::conjugate_cc::make());
      gr::blocks::multiply_cc::sptr    delay_corr(gr::blocks::multiply_cc::make());
      gr::filter::fir_filter_ccf::sptr delay_ma(gr::filter::fir_filter_ccf::make(1, std::vector(fft_len/2, use_even_carriers ?
      gr::blocks::complex_to_mag_squared::sptr delay_magsquare(gr::blocks::complex_to_mag_squared::make());
      gr::blocks::divide_ff::sptr      delay_normalize(gr::blocks::divide_ff::make());

      gr::blocks::complex_to_mag_squared::sptr normalizer_magsquare(gr::blocks::complex_to_mag_squared::make());
gr::filter::fir_filter_ccf::sptr delay_ma(gr::filter::fir_filter_ccf::make(1, std::vector(fft_len/2, use_even_carriers ? 1.0 :
      gr::blocks::complex_to_mag_squared::sptr delay_magsquare(gr::blocks::complex_to_mag_squared::make());
      gr::blocks::divide_ff::sptr      delay_normalize(gr::blocks::divide_ff::make());

      gr::blocks::complex_to_mag_squared::sptr normalizer_magsquare(gr::blocks::complex_to_mag_squared::make());
      gr::filter::fir_filter_fff::sptr         normalizer_ma(gr::filter::fir_filter_fff::make(1, std::vector(fft_len, 0.5)));
      gr::blocks::multiply_ff::sptr            normalizer_square(gr::blocks::multiply_ff::make());

      gr::blocks::complex_to_arg::sptr         peak_to_angle(gr::blocks::complex_to_arg::make());
      gr::blocks::sample_and_hold_ff::sptr     sample_and_hold(gr::blocks::sample_and_hold_ff::make());

      gr::blocks::plateau_detector_fb::sptr    plateau_detector(gr::blocks::plateau_detector_fb::make(cp_len));

      // Delay Path
      connect(self(),               0, delay,                0);
      connect(delay,                0, delay_conjugate,      0);
      connect(delay_conjugate,      0, delay_corr,           1);
      connect(self(),               0, delay_corr,           0);
      connect(delay_corr,           0, delay_ma,             0);
      connect(delay_ma,             0, delay_magsquare,      0);
      connect(delay_magsquare,      0, delay_normalize,      0);
      // Energy Path
      connect(self(),               0, normalizer_magsquare, 0);
      connect(normalizer_magsquare, 0, normalizer_ma,        0);
      connect(normalizer_ma,        0, normalizer_square,    0);
      connect(normalizer_ma,        0, normalizer_square,    1);
      connect(normalizer_square,    0, delay_normalize,      1);
      // Fine frequency estimate (output 0)
      connect(delay_ma,             0, peak_to_angle,        0);
      connect(peak_to_angle,        0, sample_and_hold,      0);
      connect(sample_and_hold,      0, self(),               0);
      // Peak detect (output 1)
      connect(delay_normalize,      0, plateau_detector,     0);
      connect(plateau_detector,     0, sample_and_hold,      1);
```

```
    connect(plateau_detector,     0, self(),              1);
#ifdef SYNC_ADD_DEBUG_OUTPUT
    // Debugging: timing metric (output 2)
    connect(delay_normalize,      0, self(),              2);
#endif
  }
```

Let's understand the code piece wise. The hierarchical block is C++ is instantiated as follows:

```
ofdm_sync_sc_cfb_impl::ofdm_sync_sc_cfb_impl(int fft_len, int cp_len, bool use_even_carriers)
      : hier_block2 ("ofdm_sync_sc_cfb",
                     io_signature::make(1, 1, sizeof (gr_complex)),
#ifndef SYNC_ADD_DEBUG_OUTPUT
                 io_signature::make2(2, 2, sizeof (float), sizeof (unsigned char)))
#else
                 io_signature::make3(3, 3, sizeof (float), sizeof (unsigned char), sizeof (float)))
```

where `ofdm_sync_sc_cfb_impl::ofdm_sync_sc_cfb_impl` is the constructor with parameters `int fft_len, int cp_len, bool use_even_carriers` and `hier_block2` is the base class. The block name `"ofdm_sync_sc_cfb"` is defined following the GNU Radio block naming style.

`io_signature::make(1, 1, sizeof (gr_complex))` defines my input items and the output items are either `io_signature::make2(2, 2, sizeof (float), sizeof (unsigned char))` or `io_signature::make3(3, 3, sizeof (float), sizeof (unsigned char), sizeof (float))`

depending on the preprocessor directive `SYNC_ADD_DEBUG_OUTPUT`.

The individual blocks inside the `ofdm_sync_sc_cfb` block are defined as follows:

```
gr::blocks::complex_to_mag_squared::sptr normalizer_magsquare(gr::blocks::complex_to_mag_squared::make());
```

Finally the individual blocks are connected using:

```
connect(normalizer_magsquare, 0, normalizer_ma, 0);
```

Retrieved from "https://wiki.gnuradio.org/index.php?title=Guided_Tutorial_GNU_Radio_in_C%2B%2B&oldid=7141"

Category: Guided Tutorials

Tutorials > Guided Tutorials

- This page was last modified on 31 May 2020, at 14:12.
-