

Tutorial 11: Writing A Signal Processing Block for GNU Radio

– Part II

Dawei Shen*

June 15, 2005

Abstract

This article continues our discussion on building a signal processing block for GNU Radio, focusing on how to glue C++ and Python all together. Some miscellaneous tips useful in GNU Radio will also be introduced.

1 Overview

In last tutorial, we have mentioned that writing a new signal processing block involves creating 3 files: The `.h` and `.cc` files that define the new block class and the `.i` file that tells SWIG how to generate the glue that binds the class into Python. We have finished `howto_square_ff.h` and `howto_square_ff.cc`, so this time we will complete the `.i` file. After that, we should arrange the directory layout properly, putting these files into correct places. Finally, we need a `Makefile.am` to get all these stuffs built.

We all feel disgusted about the huge amount of hacking in makefile. Fortunately, we use the GNU autotools to help us reduce the complexity. A very good news is that GNU Radio has provided the boilerplate that can be used pretty much as-is.

When you read this tutorial, it's suggested you have downloaded the tarball `gr-howto-write-a-block`, and take a look at the files that we will introduce.

2 The SWIG file: `how.i`

SWIG, the Simplified Wrapper and Interface Generator, is used to generate the glue that allows our code to be used from Python. We need to write the SWIG `.i` file to tell SWIG the gluing guidelines.

A `.i` file can be treated as a pared-down version of the `.h` file, plus a bit of magic that has Python work with the `boost::shared_ptr`. To reduce the code bloat, we only declare methods that we'll want to access from Python.

We're going to call the `.i` file `howto.i`, and use it to hold the SWIG declarations for all classes with the prefix `'howto_'` that will be accessible from Python. `'howto'` then corresponds to a package name in Python. The `.i` file is quite small:

```
1 /* -*- c++ -*- */
```

*The author is affiliated with Networking Communications and Information Processing (NCIP) Laboratory, Department of Electrical Engineering, University of Notre Dame. Welcome to send me your comments or inquiries. Email: dshen@nd.edu

```

2
3 %feature("autodoc", "1");          // generate python docstrings
4
5 %include "exception.i"
6 %import "gnuradio.i"              // the common stuff
7
8 %{
9 #include "gnuradio_swig_bug_workaround.h" // mandatory bug fix
10 #include "howto_square_ff.h"         // the header file
11 #include <stdexcept>
12 %}
13 // -----
14 /*
15 * GR_SWIG_BLOCK_MAGIC does some behind-the-scenes magic so we can
16 * access howto_square_ff from Python as howto.square_ff()
17 * First argument 'howto' is the package prefix.
18 * Second argument 'square_ff' is the name of the class minus the prefix.
19 */
20 GR_SWIG_BLOCK_MAGIC(howto,square_ff);
21
22 /*
23 * howto_make_square_ff is the friend function of class howto_square_ff
24 * It's the public interface for creating new instances
25 */
26 howto_square_ff_sptr howto_make_square_ff ();
27
28 /*class declaration*/
29 class howto_square_ff : public gr_block
30 {
31 private:
32     howto_square_ff ();          //private constructor
33 };

```

Just use this file as a template and ignore the details involved. Leave the red part as it is and replace the blue part with appropriate contents following the same formats. The explanation has been added into the code as comments. Please read them carefully.

Note that `GR_SWIG_BLOCK_MAGIC` does some ‘magics’ so that we can access `howto_square_ff` from Python as `howto.square_ff()`. From Python’s point of view, `howto` is a package, and `square_ff()` becomes a function defined in `howto`. Calling this function will return a smart pointer `howto_square_ff_sptr` pointing to a new instance of `howto_square_ff`.

3 Directory layout

Next, we need to organize these files properly, placing them into right positions. Table 1, ‘Directory Layout’ shows the directory layout and common files we’ll be using. After renaming the `topdir` directory, we can use it in our projects too.

To reduce the amount of our work, it’s a good idea to copy the entire folder as our own workspace and modify the files according to our own need. Again, if you feel uncomfortable with the annoying hacking in `makefile`, just take those files as templates and replace new contents at right places.

In Table 1, the mark ‘**u**’ means ‘unchanged’. Just leave them as they were. The files in dark orchid marked with ‘**s**’ need to be ‘slightly modified’, with maybe only one or two items changed. The only file marked with ‘**m**’, `/src/lib/Makefile.am`, is the real work for us. We will show these files later.

Table 1: Directory Layout

File/Dir Name	Comments
<code>topdir/Makefile.am</code> (u)	Top level Makefile.am
<code>topdir/Makefile.common</code> (u)	Common fragment included in sub-Makefiles
<code>topdir/bootstrap</code> (u)	Runs autoconf, automake, libtool first time through
<code>topdir/config</code> (u)	Directory of m4 macros used by configure.ac
<code>topdir/configure.ac</code> (s)	Input to autoconf
<code>topdir/src</code>	
<code>topdir/src/Makefile.am</code> (u)	
<code>topdir/src/lib</code>	C++ code goes here, including <code>.h</code> , <code>.cc</code> and <code>.i</code> files
<code>topdir/src/lib/Makefile.am</code> (m)	
<code>topdir/src/python</code>	Python code goes here, for testing and 'make check'
<code>topdir/src/python/Makefile.am</code> (s)	
<code>topdir/src/python/run_tests</code> (u)	Script to run tests in the build tree

We need to emphasize a couple of things before we move on.

3.1 Necessary autotools for building

Let's talk a bit about the overall building environment and explain the functions of the files listed in the directory layout that we'll be using.

To reduce the amount of makefile hacking that we have to do, and to facilitate portability across a variety of systems, we use the GNU `autoconf`, `automake`, and `libtool` tools. These are collectively referred to as the `autotools`, and once you get over the initial shock, they will become your friends. The good news is that we can use the files listed above as boilerplates, without too much changing.

3.1.1 Automake

`Automake` and `configure` work together to generate GNU compliant Makefiles from a much higher level description contained in the corresponding `Makefile.am` file. `Makefile.am` specifies the libraries and programs to build and the source files that compose each. `Automake` reads `Makefile.am` and produces `Makefile.in`. `Configure` reads `Makefile.in` and produces `Makefile`. The resulting Makefile contains a zillion rules that do the right right thing to build, check and install your code. It is not uncommon for the the resulting Makefile to be 5 or 6 times larger than `Makefile.am`.

3.1.2 Autoconf

`Autoconf` reads `configure.ac` and produces the configure shell script. `configure` automatically tests for features of the underlying system and sets a bunch of variables and defines that can be used in the Makefiles and your C++ code to conditionalize the build. If features are required but not found, `configure` will output an error message and stop.

3.1.3 Libtool

`libtool` works behind the scenes and provides the magic to construct shared libraries on a wide variety of systems.

3.2 Build tree vs. Install tree

The build tree is everything from `topdir` (the one containing `configure.ac`) down. The directory layout in Table 1 actually forms the building tree. The path to the install tree is

```
prefix/lib/python2.x/site-packages
```

where `prefix` is the `--prefix` argument to `configure` (default `/usr/local`) and `2.x` is the installed version of python. A typical path to the install tree is `/usr/local/lib/python2.3/site-packages`.

We normally set our `PYTHONPATH` environment variable to point at the install tree, and do this in `~/.bash_profile`. This allows Python to access all the standard python libraries, plus our locally installed stuff like GNU Radio.

After we finish writing our own signal processing block code, we should follow the same way as we install GNU Radio, using:

```
$ ./bootstrap
$ make
$ make check
$ make install
```

Note that to run `./bootstrap` successfully, you have to install the autotools mentioned above: `automake`, `autoconf` and `libtools`. `bootstrap` is just a script calling these tools to process the `configure` and `make` files.

3.3 Make check

We write our applications such that they access the code and libraries in the install tree. On the other hand, we want our test code to run on the build tree, where we can detect problems before installation. That's exactly what `'make check'` does.

To do that, we need to write a piece of Python testing code with the prefix `'qa_'`, and put it in `/src/python`. `'qa'` stands for 'Quality Assurance'. In this Python script, we can test the block we have just created using `'howto.square_ff()'`, and see whether it works correctly.

Then we can use `make check` to run our tests. `Make check` invokes the `/src/python/run_tests` shell script which sets up the `PYTHONPATH` environment variable so that our tests use the build tree versions of our code and libraries. It then runs all files which have names of the form `qa_*.py` and reports the overall success or failure. There is quite a bit of behind-the-scenes action required to use the non-installed versions of our code, you can look at `run_tests` for a cheap thrill.

4 Modifying configuration and make files

4.1 /src/lib/Makefile.am

This is the file that requires the most work. Here is the 'template', which is also available at:

```
/gr-howto-write-a-block/src/lib/Makefile.am
```

```
1 include $(top_srcdir)/Makefile.common
```

```

2
3 # Install this stuff so that it ends up as the gnuradio.howto module
4 # This usually ends up at:
5 #   ${prefix}/lib/python${python_version}/site-packages/gnuradio
6
7 ourpythondir = $(grpythondir)
8 ourlibdir    = $(grpyexecdir)
9
10 INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)
11
12 SWIGCPPPYTHONARGS = -noruntime -c++ -python $(PYTHON_CPPFLAGS) \
13     -I$(swigincludedir) -I$(grincludedir)
14
15 ALL_IFILES =
16     $(LOCAL_IFILES)
17     $(NON_LOCAL_IFILES)
18
19 NON_LOCAL_IFILES =
20     $(GNURADIO_CORE_INCLUDEDIR)/swig/gnuradio.i
21
22 # The .i file comes here
23 LOCAL_IFILES =
24     howto.i
25
26 # These files are built and by SWIG.
27 # The first is the C++ glue.
28 # The second is the python wrapper that loads the _howto shared library
29 # and knows how to call our extensions.
30
31 BUILT_SOURCES =
32     howto.cc
33     howto.py
34
35 # This gets howto.py installed in the right place
36 ourpython_PYTHON =
37     howto.py
38
39 ourlib_LTLIBRARIES = _howto.la
40
41 # These are the source files that go into the shared library
42 _howto_la_SOURCES =
43     howto.cc
44     howto_square_ff.cc
45
46 # magic flags
47 _howto_la_LDFLAGS = -module -avoid-version
48
49 # link the library against some common swig runtime code and the
50 # c++ standard library
51 _howto_la_LIBADD =
52     -lgrswigrunpy
53     -lstdc++
54
55 howto.cc howto.py: howto.i $(ALL_IFILES)
56     $(SWIG) $(SWIGCPPPYTHONARGS) -module howto -o howto.cc $<
57
58 # These headers get installed in ${prefix}/include/gnuradio

```

```

59 grinclude_HEADERS =          \
60     howto_square_ff.h
61
62 # These swig headers get installed in ${prefix}/include/gnuradio/swig
63 swiginclude_HEADERS =          \
64     $(LOCAL_IFILES)
65
66 MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc

```

This `Makefile.am` file gets everything built and will build a shared library from the source. It also incorporates **SWIG** and add the glue it generates to the shared library. Pay close attention to the statements including ‘`howto`’, replace them with appropriate contents when you build other blocks.

4.2 `topdir/configure.ac`

Only the first several lines need to be modified slightly:

```

1 AC_INIT
2 AC_PREREQ(2.57)
3 AC_CONFIG_SRCDIR([src/lib/howto.i])
4 AM_CONFIG_HEADER(config.h)
5 AC_CANONICAL_TARGET([])
6 AM_INIT_AUTOMAKE(gr-howto-write-a-block,0.3)

```

Modify the blue parts only.

4.3 `/src/python/Makefile.am`

A very simple file.

```

1 include $(top_srcdir)/Makefile.common
2
3 EXTRA_DIST = run_tests.in
4
5 TESTS =          \
6     run_tests
7
8 noinst_PYTHON =          \
9     qa_howto.py

```

Any files listed in ‘`noinst_PYTHON`’ will not be compiled. Just list all your Python testing files here. `qa_howto.py` is the testing Python script for our example, we will talk about it later.

5 Python testing script: `qa_howto.py`

Now let’s write a testing Python script, which should goes to `/src/python/`. The name of the file should starts with `_qa`. The file will be executed when ‘**make check**’ is performed.

```

1  #!/usr/bin/env python
2
3  from gnuradio import gr, gr_unittest
4  import howto
5
6  class qa_howto (gr_unittest.TestCase):
7
8      def setUp (self):
9          self.fg = gr.flow_graph ()
10
11     def tearDown (self):
12         self.fg = None
13
14     def test_001_square_ff (self):
15         src_data = (-3, 4, -5.5, 2, 3)
16         expected_result = (9, 16, 30.25, 4, 9)
17         src = gr.vector_source_f (src_data)
18         sqr = howto.square_ff ()
19         dst = gr.vector_sink_f ()
20         self.fg.connect (src, sqr)
21         self.fg.connect (sqr, dst)
22         self.fg.run ()
23         result_data = dst.data ()
24         self.assertEqual (expected_result, result_data, 6)
25
26 if __name__ == '__main__':
27     gr_unittest.main ()

```

`gr_unittest` is an extension to the standard python module `unittest`. `gr_unittest` adds support for checking approximate equality of tuples of float and complex numbers. `unittest` uses Python's reflection mechanism to find all methods that start with `test_` and runs them. `unittest` wraps each call to `test_*` with matching calls to `setUp` and `tearDown`. See the python [unittest](#) documentation and the Python file `sitepackages/gnuradio/gr_unittest.py` for more details.

When we run the test, `gr_unittest.main` is going to invoke `setUp`, `test_001_square_ff`, and `tearDown`.

`test_001_square_ff` builds a small graph that contains three nodes. `gr.vector_source_f(src_data)` will source the elements of `src_data` and then say that it's finished. `howto.square_ff` is the block we're testing. `gr.vector_sink_f` gathers the output of `howto.square_ff`.

The `run` method runs the graph until all the blocks indicate they are finished. Finally, we check that the result of executing `square_ff` on `src_data` matches what we expect.

This `qa_howto.py` file provides a very good framework for writing test scripts. We can always take the advantage of `gr_unittest` to design our own test Python files.

6 We are done!

OK, that's everything we need! `./bootstrap`, `configure` and `make`, everything should be built successfully. We get a few warnings, but that's ok. Changing the directory to `src/python`, and try **make check**, we should be able to pass the test.

```
sachi@dshen:~/gr-howto-write-a-block/src/python$ make check
```

```
make check-TESTS
make[1]: Entering directory '/home/sachi/gr-howto-write-a-block/src/python'
..
-----
Ran 2 tests in 0.025s

OK
PASS: run_tests
=====
All 1 tests passed
=====
make[1]: Leaving directory '/home/sachi/gr-howto-write-a-block/src/python'
```

Excellent! We have a new block working!

7 Conclusion

Let's take another break. At this point, you should be able to write your own blocks and know how to test it and build it into the GNU Radio tree. In the next tutorial, we will introduce some subclasses of `gr_block` and visit our `howto_square_ff()` example again.

References

- [1] Eric Blossom, **How to Write a Signal Processing Block**,
<http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html>
- [2] **Python on-line tutorials**, <http://www.python.org/doc/current/tut/>
- [3] Eric Blossom, **Exploring GNU Radio**,
<http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>