ARMY RESEARCH LABORATORY

Developing Signal Processing Blocks for Software-defined Radios

by Gunjan Verma and Paul Yu

ARL-TR-5897 **January 2012**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-5897 **January 2012**

Developing Signal Processing Blocks for Software-defined Radios

Gunjan Verma and Paul Yu
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
January 2012	Final	

4. TITLE AND SUBTITLE
Developing Signal Processing Blocks for Software-defined Radios

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Gunjan Verma and Paul Yu

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-CIN-T
2800 Powder Mill Road
Adelphi, MD 20783-1197

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-5897

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Software-defined radios (SDRs) provide researchers with a powerful and flexible wireless communications experimentation platform. GNU Radio is the most popular open-source software toolkit for deploying SDRs, and is frequently used with the Universal Software Radio Peripheral (USRP). After establishing a USRP testbed, the researcher will need to implement new signal processing algorithms or modify existing ones. This document describes this process, highlighting those details that have received minimal attention in the existing documentation.

15. SUBJECT TERMS
Software radios, CHU Radio, USRP Ubuntu

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	44	Gunjan Verma
Unclassified	Unclassified	Unclassified			19b. TELEPHONE NUMBER (*Include area code*) (301) 394-3102

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Tables

INTENTIONALLY LEFT BLANK

1. Introduction

Software-defined radios (SDRs) provide researchers with a powerful and flexible wireless communications experimentation platform. GNU Radio is the most popular open-source software toolkit for deploying. Every SDR is comprised of software and hardware. In this document, we consider GNU Radio software coupled with Universal Software Radio Peripheral (USRP) hardware. In GNU Radio, C++ ***blocks*** perform specific signal processing processing tasks, while Python ***applications*** connect the blocks together to form a functional software radio. For example, a basic transmitter can be implemented by using Python to connect the following C++ blocks (which already exist in the GNU Radio software library) together: modulator, mixer, and amplifier.

Each block specifies its input and output requirements, both in number and type. For example, the `gr_add_cc` block adds two complex input streams and copies the results onto one complex output stream. Blocks are generally implemented in C++ for computational efficiency, but other possibilities exist (see below).

After writing a new block, a process is needed to expose these C++ blocks for use by Python scripts. GNU Radio uses the Simplified Wrapper and Interface Generator (SWIG), to generate the necessary components to make C++ blocks accessible from Python.

From the standpoint of Python applications, each block consumes its input stream(s), performs a specific task, and generates output stream(s). As long as the connections between blocks are compatible, there is no restriction to how many blocks can be chained together. A single output stream can connect to multiple input streams, but multiple outputs cannot connect to a single input due to ambiguity. A multiplexer can be used in such a situation by interleaving many inputs onto a single output.

In summary, the stages of block creation in GNU Radio are the following:

1. Implementation of blocks (C++), the (.h, .cc) files
2. Creation of SWIG interfaces between C++/Python, the (.i) file
3. Installation of blocks into a shared library
4. Usage of blocks in an application (Python), the (.py) file

In this report, we detail steps 1–4. This report is an updated and expanded presentation of the material found in
http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html.

2. Implementation of Blocks

Before diving into block implementation, we first introduce the naming conventions of GNU Radio in section 2.1. Section 2.2 introduces the most commonly used data types, and section 2.3 steps through the essential elements of block creation by using the illustrative example of `gr_block`.

2.1 Naming Conventions

There are several strongly followed conventions in GNU Radio, and a familiarity with these expedites code writing and understanding.

- All words in identifiers are separated by an underscore, e.g., `gr_vector_int`.
- All types in the GNU Radio package are preceded by `gr`, e.g., `gr_float`.
- All class variables are preceded with `d_`, e.g., `d_min_streams`.
- Each classes is implemented in a separate file, e.g., class `gr_magic` is implemented in `gr_magic.cc` with the header file `gr_magic.h`.
- All signal processing blocks contain their input and output types in their suffixes, e.g., `gr_fft_vcc` requires complex inputs and complex outputs. The major types are float (f), complex (c), short (s), integer (i). Any type may be vectorized (v).

2.2 Data Types

GNU Radio type defines the most commonly used data types to a set of names. The main purpose of this is to create a common set of conventions for naming of data types. The list is as follows:

```
typedef std::complex<float>          gr_complex;
typedef std::complex<double>         gr_complexd;
typedef std::vector<int>             gr_vector_int;
typedef std::vector<float>           gr_vector_float;
```

```
typedef std::vector<double>             gr_vector_double;
typedef std::vector<void *>             gr_vector_void_star;
typedef std::vector<const void *>       gr_vector_const_void_star;
typedef short                           gr_int16;
typedef int                             gr_int32;
typedef unsigned short                  gr_uint16;
typedef unsigned int                    gr_uint32;
```

2.2.1 Block Signatures

A block signature is simply a specification of the data types that enter and exit a signal processing block. In list 1, we can examine `gr_io_signature.h` for more detail.

Listing 1. gr_io_signature.h

```
1  class gr_io_signature {
2    int                     d_min_streams;
3    int                     d_max_streams;
4    std::vector<int>        d_sizeof_stream_item;
5
6    gr_io_signature(int min_streams, int max_streams,
7                    const std::vector<int> &sizeof_stream_items);
8
9    friend gr_io_signature_sptr
10   gr_make_io_signaturev(int min_streams,
11                         int max_streams,
12                         const std::vector<int> &sizeof_stream_items);
13
14 public:
15
16   static const int IO_INFINITE = -1;
17
18   ~gr_io_signature ();
19
20   int min_streams () const { return d_min_streams; }
21   int max_streams () const { return d_max_streams; }
22   int sizeof_stream_item (int index) const;
23   std::vector<int> sizeof_stream_items() const;
24 };
```

It is important to realize that our block has two signatures, one for the input interface and one for the output interface. The header file makes it clear that, for a given interface, `gr_io_signature` defines the minimum and maximum number of streams flowing through that interface, as well as the number of bytes in a single element of the stream. Recall that Python is used to connect multiple signal processing blocks together. The main purpose of signatures is so Python can raise an error for improper connections.

The following are examples of improper connections:

- Too {many/few} {input/output} connections for a block
- Type mismatch, e.g., `gr_complex` output connected to `gr_int16` input

2.2.2 Boost Pointers

GNU Radio uses Boost smart pointers instead of regular C++ pointers. Boost is a high-quality software library with many extensions to the basic C++ language. For our purposes, Boost provides a smart implementation of C++ pointers that offers garbage collection, i.e., it deletes dynamically allocated objects when they are no longer needed. This simplifies our implementation efforts and improves block performance. There are actually many different types of smart pointers, but GNU radio uses just one of them, called a `shared_ptr`, which is used specifically when our dynamically allocated object has ownership shared by several pointers.

In order to declare a regular C++ pointer to an object of type `gr_io_signature`, we would use the followig command:

```
gr_io_signature* ptr;
```

Whereas with Boost, we would use this command:

```
typedef boost::shared_ptr<gr_io_signature> gr_io_signature_sptr;
gr_io_signature_sptr ptr;
```

to declare a Boost shared pointer.

As shown in the above code, GNU Radio uses the convention of type defining Boost smart pointers to an object of type `X` as `X_sptr`. This format makes it explicit to the user that `X_sptr` is a Boost smart pointer.

2.3 Case Study: gr_block

GNU Radio makes extensive use of the notion of "inheritance," an object oriented (OO) programming technique. For us, this simply means that every signal processing block is a specialization of a general, high-level block, which GNU Radio calls `gr_block`. Our task is to fill in the details of `gr_block` (referred to in OO-speak as "deriving from the base

class") to create our own custom block. It is prudent to begin our study of writing a new block by first examining `gr_block.h`.

The class `gr_block` is itself derived from the class `gr_basic_block.h`. We consider a few of the fields that are of particular interest to programmer and discuss the fields inherited from `gr_basic_block.h` and defined `gr_block.h` (lists 2 and 3). The entire `gr_block.h` file is shown in appendix A.

Listing 2. gr_basic_block.h

```
70      std::string             d_name;
71      gr_io_signature_sptr    d_input_signature;
72      gr_io_signature_sptr    d_output_signature;
73      long                    d_unique_id;
```

Listing 3. gr_block.h

```
229  private:
230
231   int                     d_output_multiple;
232   double                  d_relative_rate;        // approx output_rate /
          input_rate
```

The fields `d_name` and `d_unique_id` are unique identifiers (text and numeral, respectively) for the block and can be used for debugging. The `d_output_multiple` and `d_relative_rate` fields inform the schedule of the block's rate of data consumption and generation (see sections 2.3.3 and 2.3.4).

Note that `d_input_signature`, `d_output_signature`, and `d_detail` are all Boost smart pointers, the former pointing to `gr_io_signature` objects, the latter to a `gr_block_detail` object. The comments above highlight the purpose of the various fields; we explain in more detail in what follows.

As seen in list 4, `gr_block` has the following important functions.

Listing 4. gr_block.h

```
 82   void  set_history (unsigned history) { d_history = history; }
105   virtual void forecast (int noutput_items,
106                          gr_vector_int &ninput_items_required);
122   virtual int general_work (int noutput_items,
123                             gr_vector_int &ninput_items,
124                             gr_vector_const_void_star &input_items,
125                             gr_vector_void_star &output_items) = 0;
157   void consume (int which_input, int how_many_items);
```

In the remainder of this section, we detail each of these functions. It is useful to think of the process of writing a new block as a "two-way street" between our block and the GNU

Radio internals, collectively referred to herein as the ***scheduler***. The scheduler gives us data from the USRP, on which our block performs signal processing. In turn, our block tells the scheduler how much processing we've done and how much more input we need to produce more output, so the scheduler knows what data it no longer needs to store, how much buffer memory to allocate, when to schedule our block to execute next, etc. This, in turn, determines when the scheduler will invoke our block next and with how much input.

2.3.1 Function: general_work()

The `general_work()` function plays a central role in new block creation. It implements the process of converting the input stream(s) to the output stream(s). Table 1 explains the purpose of the arguments to this function.

Table 1. Arguments of general_work().

Argument	Purpose
noutput_items	Number of output items to write on each output stream
ninput_items	Number of input items available on each input stream
input_items	Vector of pointers to elements of the input stream(s), i.e., element i of this vector points to the i^{th} input stream
output_items	Vector of pointers to elements of the output stream(s), i.e., element i of this vector points to the i^{th} output stream

Recall that we stated earlier that a signal processing block may have multiple input and/or output streams. So, `ninput_items` is vector whose i^{th} element is the number of available items on the i^{th} input stream. However, `noutput_items` is a scalar, not a vector, because GNU Radio implementation forces the number of output items to write on each output stream to be the same. The returned value of `general_work()` is the number of items actually written to each output stream, or -1 on end of file (EOF).

To create a block, we simply define how to create `output_items` from `input_items`, assuming that all parameters are provided to us. That is, ***we implement the signal processing algorithm*** in this method. The scheduler invokes the concrete implementation of `general_work` with the appropriate parameters. We do not have to explicitly invoke `general_work`; we only need to define it.

After we have defined `general_work` for our custom signal processing block, we need to invoke the `consume()` function to indicate to the scheduler how many items (`how_many_items`) have been processed on each (`which_input`) input stream. Recall that the scheduler is providing us all the appropriate parameters for us to write our own block; we need to provide feedback to the scheduler so it knows which elements have been used,

so it can mark appropriate memory for deletion or reuse, and update pointers to point to new data. This feedback of our signal processing progress is provided to the scheduler via the consume function.

2.3.2 Functions: forecast() and set_history()

The `forecast()` function is our way of telling the scheduler our estimate of the number of input elements that will be needed to create an output element. For example, a decimating filter of order 5 requires five inputs to produce one output. The key argument is `ninput_items_required`, which is a vector specifying the number of input items required on each input stream to produce `noutput_items` number of outputs on each output stream. In some cases, like the decimating filter of specified order, we may know this number exactly. In other cases, we may need to estimate it. When the scheduler determines it is ready to handle `noutput_items` more items on the output streams, it invokes the `forecast` function to determine whether or not we have enough input items to call `general_work`. For example, an interpolator will produce multiple outputs for a single input, while a decimator will produce a single output for multiple inputs. If we have a 10-to-1 decimator but only 9 inputs are available, the scheduler will not call `general_work` if the `forecast` function is correctly implemented.

There is an important distinction to make here. Another common requirement, such as for a moving average filter that averages the five most recent inputs to produce a single output, is the need to process multiple input samples to yield a single output sample. It may seem as if our forecast function should specify this. However, in this case, while the moving average filter uses five inputs to produce one output, it does not require five *new* inputs; it still only consumes a single *new* input to produce a single *new* output. So, our `forecast` function, in this case, would still call for a one-to-one relation of `noutput_items` to `ninput_items_required`. In this case, the fact that we need the five most recent inputs would be specified to GNU Radio via the `set_history(5)` function call.

2.3.3 Field: d_output_multiple

By now, we have seen that the GNU Radio scheduler is responsible for invoking `general_work` and `forecast`. The `forecast()` function allows us to signal to the scheduler to invoke our `general_work` function only when a sufficient number of input elements are in the input buffer. But we have not seen any such mechanism to control the number of outputs being produced. Recall the argument `noutput_items` in the `forecast` function. It it specified by the scheduler and contains how many output items to produce on each stream. While we cannot directly set this value (it is under the scheduler's control), there is a variable `d_output_multiple` that tells the scheduler that the value of

noutput_items must be an integer multiple of d_output_multiple. In other words, the scheduler only invokes forecast() and general_work() if noutput_items is an integer multiple of d_output_multiple. The default value of d_output_multiple is 1. Suppose, for instance, we are interested in generating output elements *only* in 64-element chunks. By setting d_output_multiple to 64, we can achieve this, but note that we may also get any multiple of 64, such as 128 or 192, instead.

The following functions allow us to set and get the value of d_output_multiple:

```
void    gr_block::set_output_multiple (int multiple);
int     output_multiple ();
```

2.3.4 Field: d_relative_rate

Recall our description of block creation as involving a "two-way communication" with the scheduler. The d_relative_rate field is the way we tell the scheduler the approximate ratio of output rate to input rate at which we expect our signal processing algorithm to operate. The key purpose of d_relative_rate is to allow the scheduler to optimize its use memory and timings of invocation of general_work. For many blocks, d_relative_rate is 1.0 (the default value), but decimators will have a value less than 1.0 and interpolators greater than 1.0.

The functions used to set and get the value of d_relative_rate are given below:

```
void    gr_block::set_relative_rate (double relative_rate);
double  relative_rate ();
```

3. Creation of SWIG Interfaces

In what follows, we strongly urge the reader to download the file gr-howto-write-a-block-3.3.0.tar.gz from ftp.gnu.org/gnu/gnuradio/ and extract the archive. This archive contains sample code related to creating a new block, which we refer to.

3.1 Naming Conventions

Before getting into the details of block creation, let us start with a note about some important naming conventions.

3.1.1 Block Names

After we create our new block, the only way we can use it in GNU Radio is to create a Python script, which loads the package/module containing the block, and then connect our block into a GNU Radio flowgraph, as usual. This would involve Python code resembling the following:

```
from package_name import module_name
...
nb = module_name.block_name()
...
```

There is a key coupling between the module and block names that we invoke in Python, and the names used in coding blocks in C++. Namely, GNU Radio expects that all C++ source and header files are in the form [module_name]_[block_name].h and [module_name]_[block_name].cc . That is, if we decided to name our C++ class `newModule_newBlock`, then GNU Radio's build system would make our block available from Python in module "newModule" and with block name "newBlock". So while in theory there is no need for such a coupling of naming schemes, in practice, such a coupling does exist.

3.1.2 Boost Pointers

We have mentioned earlier that all pointers to GNU Radio block objects must use Boost shared pointers not "regular" C++ pointers. In other words, if we create a new C++ signal processing block called "`newModule_newBlock`", then GNU radio's internal implementation will not work if we use a pointer to newBlock_newFunction in our code. In other words, the command

```
newModule_newBlock* nb = new newModule_newBlock()
```

is not permitted. This is enforced by making all block constructors private and ensures that a regular C++ pointer can never point to a block object. But if the constructor is private, how do we create new instance of our block? After all, we need some sort of public interface for creating new block instances. The solution is to declare a "friend function," which acts as a surrogate constructor. This is achieved by first declaring a friend function of the class, so it has access to all private members, including the private constructor. This friend function invokes the private constructor and returns a smart pointer to it. Second, we invoke this friend function every time we want to construct a new object.

Suppose the name of our new signal processing block is `newModule_newBlock_cc`. Then, we would create a file `newModule_newBlock_cc.cc`, in which we would include the following function declaration:

```
typedef boost::shared_ptr<newModule_newBlock_cc> newModule_newBlock_cc_sptr;
friend newNodule_newBlock_cc_sptr newModule_make_newBlock_cc()
```

Now, the function `newModule_newBlock_make_cc()` has access to private members of the class `newModule_newBlock_cc`. So from within this function, we call the private constructor of `newModule_newBlock_cc`, in order to create a new instance of our block. The final step is to cast the returned pointer's data type from a raw C++ pointer to a smart pointer

```
newNodule_newBlock_cc_sptr  newModule_make_newBlock_cc() () {
    return newNodule_newBlock_cc_sptr (new newModule_newBlock_cc());
}
```

The private constructor (which we cannot invoke directly), on the other hand, would look something like this.

```
newNodule_newBlock_cc ()   {
          gr_block (‘‘newBlock_cc",
          gr_make_io_signature (1, 1, sizeof (gr_complex)),
          gr_make_io_signature (1, 1, sizeof (gr_complex))
          )
}
```

So to summarize, the private constructor is actually creating a new gr_block object. The "friend" constructor, the public interface to the private constructor, acts as a surrogate by wrapping the new object created by the private constructor into a Boost shared pointer. This convoluted procedure guarantees that all pointers to blocks are Boost smart pointers. The public interface to creating objects is not the object constructor `newModule_newBlock_cc`, but rather the "surrogate" constructor `newModule_newBlock_make_cc`.

Then, in our code, we must create a new block object using the code

```
newModule_newBlock_cc_sptr nb = newModule_make_newBlock_cc()
```

Here is an important point: if one's block name is `newModule_newBlock_cc`, then the name of the shared pointer to this block MUST be `newModule_newBlock_cc_sptr`. Any other choice, such as `nb_nf_sptr`, would lead to the block not working properly. This has nothing to do with C++, since any valid name will work. Rather, when this C++ block is invoked from Python in a GNU Radio program, GNU Radio expects the shared pointer name to follow directly from the block name with an `_sptr` added on, or else it will complain that it cannot find the block.

Also, the surrogate constructor that creates a shared pointer to `newModule_newBlock_cc` *must* have signature

```
newModule_newBlock_cc_sptr newModule_make_newBlock_cc()
```

Note the presence of the word "make" between the newModule and newBlock words. Thus, consider the naming of shared pointers to block objects, as well as the friend functions (surrogate constructors) that create them, not as a convention *but as rule to be followed.*

3.2 SWIG Interface File

Once we have created our .cc and .h files, the next step is to create the SWIG (.i) file, so we can expose our new block to Python. SWIG is used to generate the necessary "glue," as it is often called, to allow Python and C++ to "stick" together in a complete GNU Radio application. The purpose of the .i file is to tell SWIG how it should go about creating this glue.

A .i file is very similar to a .h file in C++ in that it declares various functions. However, the .i file only declares the functions that we want to access from Python. As a result, the .i file is typically quite short in length.

We illustrate an actual .i file in list 5, called `gr_multiply_const_ff.i`.

Listing 5. gr_multiply_const_ff.i

```
1 /*
2  * GR_SWIG_BLOCK_MAGIC is a function which allows us to invoke our block
3  * gr_multiply_const_ff from Python as gr.multiply_const_cc()
4  * Its first argument, 'gr', will become the package prefix.
5  * Its second argument 'multiply_const_ff' will become the object name.
6 */
7
8 GR_SWIG_BLOCK_MAGIC(gr,multiply_const_ff)
```

```
9
10  /*
11   * gr_make_multiply_const_ff  is the surrogate constructor
12   * i.e. the friend function of class gr_multiply_const_ff
13   */
14 gr_multiply_const_ff_sptr gr_make_multiply_const_ff (float k);
15
16
17 class gr_multiply_const_ff : public gr_sync_block
18 {
19  private:
20   gr_multiply_const_ff (float k); // the "true", private constructor
21
22  public:
23   float k () const { return d_k; }
24   void set_k (float k) { d_k = k; }
25 };
```

There are some important aspects to note from the above choices of names. First, the fact that we have invoked `GR_SWIG_BLOCK_MAGIC` with parameters "`gr`" and "`multiply_const_ff`" has direct relevance to how we invoke the block from Python. Practically, this means that in Python, when we seek to invoke our blocks, we would first use the command

```
import gr
```

When we wish to instantiate our block, we would use the Python command

```
block = gr.multiply_const_ff()
```

In summary, from within Python, `gr` is a package and `multiply_const_ff` is a function within this package. The way we have created the .i file specifies the particular names that Python ascribes to the package (`gr`) and function (`multiply_const_ff`).

4. Installation of Blocks

4.1 Directory

The next step involves placing various files in the correct locations to ensure a successful build. We assume that we have finished writing all the necessary files and now make our

new blocks accessible from Python. In this section, we outline the key steps needed to build new signal processing applications in GNU radio.

A sample block is available from the GNU Radio online package archive [**?**], where each block is version numbered as X.Y.Z to correspond to the analogous version of GNU Radio. Download and unpack this block to a directory of your choice, e.g., "`newBlock`". The directory structure and significance of each folder is explained in table 2.

Table 2. Contents of newBlock directory in GNU Radio.

Directory	Contents
/home/user/newBlock	Top level Makefile, documentation
/home/user/newBlock/config	Files for GNU Autotools
/home/user/newBlock/src	Top level folder for C++ and Python files
/home/user/newBlock/src/lib	Folder for C++ source/header files

As we write our own blocks, keep in mind that all files (.h, .cc, and .i) for the new signal processing block should go in the `newBlock/src/lib` directory.

4.2 Preparing Makefile.am for Autotools

The final step before compilation is to edit the Makefile.am file (located in the previous example in the root directory, i.e., `/home/user/newBlock`). `Makefile.am` specifies which libraries to build, the source files that comprise those libraries, and the appropriate flags to use. This file contains relevant information to configure the build process that is to follow to correctly compile our code. Open this file and edit two sections. The first, shown below, identifies the name of SWIG's .i file:

```
# Specify the .i file below
LOCAL_IFILES = newModule.i
```

The next tells SWIG which files to build and what to name them for use by Python:

```
BUILT_SOURCES =                         \
newModule.cc                        \
newModule.py
```

The next set of commands ensures that our new block's Python code is installed in the proper location.

```
ourPython_PYTHON =                          \
newModule.py
ourlib_LTLIBRARIES = _newModule.la
```

The next set of commands specify which source files are included in the shared library that SWIG exposes to Python:

```
_newModule_la_SOURCES =                         \
newModule.cc                            \
newModule_newBlock_cc.cc
```

The final set of commands specify key flags to ensure that our new signal processing block shared library compiled and linked correctly against SWIG and the C++ standard library:

```
_newModule_la_LDFLAGS = -module -avoid-version
 _newModule_la_LIBADD =                         \
 -lgrswigrunpy                    \
 -lstdc++
newModule.cc newModule.py: newModule.i $(ALL_IFILES)
$(SWIG) $(SWIGCPPPYTHONARGS) -module newModule -o newModule.cc $<
grinclude_HEADERS =                         \
newModule_newBlock_cc.h
```

4.3 Installation

After we finish coding our block, we need to install it. Fortunately, this process is made easy by the included makefile in the archive downloaded earlier. With the editing of the `Makefile.am` file as above, we are now ready to build our new block. Simply use the following commands:

```
./bootstrap
./configure --prefix=prefix
make
sudo make install
sudo touch install_path/package_name/__init__.py
```

Here, "prefix" is the root of our GNU radio installation (default is `/usr/local`). Also, "install_path" is the directory where our package is being installed (default is

prefix/lib/Python/*Python-version*/site-packages, where Python-version is the version of Python being used). Finally, "package_name" is the name of the Python package under which our block will be available. This would have been specified in `Makefile.am` by us during build time, so we just enter that name here. The creation of an `__init__.py` file is necessary since Python expects every directory containing a package to have this file.

If subsequently we make changes to our code, we can repeat the above steps but omit the bootstrap and configure steps.

5. Usage of Blocks

5.1 Invoking from Python

Our final step is to use our new block from Python as part of a GNU Radio flowgraph. This is easy using the following commands:.

```
from gnuradio import newModule
.....
block = newModule.newBlock_cc ()
....
```

5.2 Debugging

The challenge of debugging our new block is that we are not executing C++ code directly. Rather, our block, comprised of C++ code, is loaded dynamically into Python and executed "through" a Python process. Therefore, the most convenient debugging option involves inserting print statements through the block source code to monitor its status during execution. For those familiar with GDB (and often, many graphical debuggers use GDB under the covers), the following code can be used:

```
from gnuradio import newModule
import os #package providing blocking function
print 'My process id is (pid = %d)' % (os.getpid(),)
raw_input ('Please attach GDB to this process ID, then hit enter: ')
# now continue using our block
block = newModule.newBlock_cc();
```

The idea of this code is simply to discover the process ID of the Python process, which invokes our new block, and then in another terminal, have GDB attach to this process ID. Now, GDB can be used as usual (to set breakpoints, watch points, etc.) When we have configured GDB as we like, we can return to the terminal executing the Python process, and hit Enter to have it proceed.

5.3 Simplifying the Build

As we have mentioned previously, there are several caveats involved in the creation of a new signal processing block. Beyond just writing the C++ code, we must create a Makefile.am file and SWIG .i file, and be careful in the naming of various files and functions so as to adhere to GNU Radio's naming rules. These steps are a "one-time cost" associated with writing a new block. Then, we have to ensure all files are placed in the correct place, and then invoke a series of commands to compile, build, and deploy our application. These latter steps are a "recurring cost," which we must incur each time we go through the debug-build-test cycle. Overall, the process of building and deploying the block can be time-consuming and error-prone. To allow us to focus on creating new signal processing blocks in C++ and avoid dealing directly with the complexities of the build process and naming rules, we have created a script in Python. After the user has written a new block in C++, this script automates the rest of the process, ensuring that all naming rules are adhered to (and renaming accordingly when necessary) and all packages are properly built and usable from Python. The script is given in the appendix B.

6. Conclusion

In this report, we have provided the details of how to create a new signal processing block using GNU Radio. We have highlighted important naming conventions; surveyed the important functions to be overridden in `gr_block`, such as `general_work` and `forecast`; and illustrated the importance of Boost smart pointers. Finally, we have discussed how to compile a block, deploy it, and invoke it from Python.

A. The gr_block.h Script

```
1  /* -*- c++ -*- */
2  /*
3   * Copyright 2004,2007,2009,2010 Free Software Foundation, Inc.
4   *
5   * This file is part of GNU Radio
6   *
7   * GNU Radio is free software; you can redistribute it and/or modify
8   * it under the terms of the GNU General Public License as published by
9   * the Free Software Foundation; either version 3, or (at your option)
10  * any later version.
11  *
12  * GNU Radio is distributed in the hope that it will be useful,
13  * but WITHOUT ANY WARRANTY; without even the implied warranty of
14  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15  * GNU General Public License for more details.
16  *
17  * You should have received a copy of the GNU General Public License
18  * along with GNU Radio; see the file COPYING.  If not, write to
19  * the Free Software Foundation, Inc., 51 Franklin Street,
20  * Boston, MA 02110-1301, USA.
21  */
22
23 #ifndef INCLUDED_GR_BLOCK_H
24 #define INCLUDED_GR_BLOCK_H
25
26 #include <gr_basic_block.h>
27
28 /*!
29  * \brief The abstract base class for all 'terminal' processing blocks.
30  * \ingroup base_blk
31  *
32  * A signal processing flow is constructed by creating a tree of
33  * hierarchical blocks, which at any level may also contain terminal nodes
34  * that actually implement signal processing functions. This is the base
35  * class for all such leaf nodes.
36
37  * Blocks have a set of input streams and output streams.  The
38  * input_signature and output_signature define the number of input
39  * streams and output streams respectively, and the type of the data
40  * items in each stream.
41  *
42  * Although blocks may consume data on each input stream at a
43  * different rate, all outputs streams must produce data at the same
44  * rate.  That rate may be different from any of the input rates.
45  *
46  * User derived blocks override two methods, forecast and general_work,
47  * to implement their signal processing behavior. forecast is called
48  * by the system scheduler to determine how many items are required on
49  * each input stream in order to produce a given number of output
50  * items.
```

```
51  *
52  * general_work is called to perform the signal processing in the block.
53  * It reads the input items and writes the output items.
54  */
55
56 class gr_block : public gr_basic_block {
57
58  public:
59
60   //! Magic return values from general_work
61   enum {
62     WORK_CALLED_PRODUCE = -2,
63     WORK_DONE = -1
64   };
65
66   enum tag_propagation_policy_t {
67     TPP_DONT = 0,
68     TPP_ALL_TO_ALL = 1,
69     TPP_ONE_TO_ONE = 2
70   };
71
72   virtual ~gr_block ();
73
74   /*!
75    * Assume block computes y_i = f(x_i, x_i-1, x_i-2, x_i-3...)
76    * History is the number of x_i's that are examined to produce one y_i.
77    * This comes in handy for FIR filters, where we use history to
78    * ensure that our input contains the appropriate "history" for the
79    * filter.   History should be equal to the number of filter taps.
80    */
81   unsigned history () const { return d_history; }
82   void  set_history (unsigned history) { d_history = history; }
83
84   /*!
85    * \brief Return true if this block has a fixed input to output rate.
86    *
87    * If true, then fixed_rate_in_to_out and fixed_rate_out_to_in may be called
       .
88    */
89   bool fixed_rate() const { return d_fixed_rate; }
90
91   // ----------------------------------------------------------------
92   //             override these to define your behavior
93   // ----------------------------------------------------------------
94
95   /*!
96    * \brief  Estimate input requirements given output request
97    *
98    * \param noutput_items           number of output items to produce
99    * \param ninput_items_required   number of input items required on each
        input stream
100   *
101   * Given a request to product \p noutput_items, estimate the number of
102   * data items required on each input stream.  The estimate doesn't have
```

```
103     * to be exact, but should be close.
104     */
105    virtual void forecast (int noutput_items,
106                           gr_vector_int &ninput_items_required);
107
108    /*!
109     * \brief compute output items from input items
110     *
111     * \param noutput_items       number of output items to write on each output
          stream
112     * \param ninput_items        number of input items available on each input
          stream
113     * \param input_items         vector of pointers to the input items, one
          entry per input stream
114     * \param output_items        vector of pointers to the output items, one
          entry per output stream
115     *
116     * \returns number of items actually written to each output stream, or -1 on
          EOF.
117     * It is OK to return a value less than noutput_items.  -1 <= return value
          <= noutput_items
118     *
119     * general_work must call consume or consume_each to indicate how many items
120     * were consumed on each input stream.
121     */
122    virtual int general_work (int noutput_items,
123                              gr_vector_int &ninput_items,
124                              gr_vector_const_void_star &input_items,
125                              gr_vector_void_star &output_items) = 0;
126
127    /*!
128     * \brief Called to enable drivers, etc for i/o devices.
129     *
130     * This allows a block to enable an associated driver to begin
131     * transfering data just before we start to execute the scheduler.
132     * The end result is that this reduces latency in the pipeline when
133     * dealing with audio devices, usrps, etc.
134     */
135    virtual bool start();
136
137    /*!
138     * \brief Called to disable drivers, etc for i/o devices.
139     */
140    virtual bool stop();
141
142    // ----------------------------------------------------------------
143
144    /*!
145     * \brief Constrain the noutput_items argument passed to forecast and
          general_work
146     *
147     * set_output_multiple causes the scheduler to ensure that the noutput_items
148     * argument passed to forecast and general_work will be an integer multiple
149     * of \param multiple  The default value of output multiple is 1.
```

```
150     */
151    void set_output_multiple (int multiple);
152    int  output_multiple () const { return d_output_multiple; }
153
154    /*!
155     * \brief Tell the scheduler \p how_many_items of input stream \p
           which_input were consumed.
156     */
157    void consume (int which_input, int how_many_items);
158
159    /*!
160     * \brief Tell the scheduler \p how_many_items were consumed on each input
           stream.
161     */
162    void consume_each (int how_many_items);
163
164    /*!
165     * \brief Tell the scheduler \p how_many_items were produced on output
           stream \p which_output.
166     *
167     * If the block's general_work method calls produce, \p general_work must
           return WORK_CALLED_PRODUCE.
168     */
169    void produce (int which_output, int how_many_items);
170
171    /*!
172     * \brief Set the approximate output rate / input rate
173     *
174     * Provide a hint to the buffer allocator and scheduler.
175     * The default relative_rate is 1.0
176     *
177     * decimators have relative_rates < 1.0
178     * interpolators have relative_rates > 1.0
179     */
180    void  set_relative_rate (double relative_rate);
181
182    /*!
183     * \brief return the approximate output rate / input rate
184     */
185    double relative_rate () const { return d_relative_rate; }
186
187    /*
188     * The following two methods provide special case info to the
189     * scheduler in the event that a block has a fixed input to output
190     * ratio.  gr_sync_block, gr_sync_decimator and gr_sync_interpolator
191     * override these.  If you're fixed rate, subclass one of those.
192     */
193    /*!
194     * \brief Given ninput samples, return number of output samples that will be
           produced.
195     * N.B. this is only defined if fixed_rate returns true.
196     * Generally speaking, you don't need to override this.
197     */
198    virtual int fixed_rate_ninput_to_noutput(int ninput);
```

```
199
200   /*!
201    * \brief Given noutput samples, return number of input samples required to
          produce noutput.
202    * N.B. this is only defined if fixed_rate returns true.
203    * Generally speaking, you don't need to override this.
204    */
205   virtual int fixed_rate_noutput_to_ninput(int noutput);
206
207   /*!
208    * \brief Return the number of items read on input stream which_input
209    */
210   uint64_t nitems_read(unsigned int which_input);
211
212   /*!
213    * \brief  Return the number of items written on output stream which_output
214    */
215   uint64_t nitems_written(unsigned int which_output);
216
217   /*!
218    * \brief Asks for the policy used by the scheduler to moved tags downstream
          .
219    */
220   tag_propagation_policy_t tag_propagation_policy();
221
222   /*!
223    * \brief Set the policy by the scheduler to determine how tags are moved
          downstream.
224    */
225   void set_tag_propagation_policy(tag_propagation_policy_t p);
226
227   // ------------------------------------------------------------------------
228
229 private:
230
231   int                     d_output_multiple;
232   double                  d_relative_rate;        // approx output_rate /
          input_rate
233   gr_block_detail_sptr    d_detail;               // implementation details
234   unsigned                d_history;
235   bool                    d_fixed_rate;
236   tag_propagation_policy_t d_tag_propagation_policy; // policy for moving tags
          downstream
237
238 protected:
239
240   gr_block (const std::string &name,
241             gr_io_signature_sptr input_signature,
242             gr_io_signature_sptr output_signature);
243
244   void set_fixed_rate(bool fixed_rate){ d_fixed_rate = fixed_rate; }
245
```

```
246
247     /*!
248      * \brief  Adds a new tag onto the given output buffer.
249      *
250      * \param which_output an integer of which output stream to attach the tag
251      * \param abs_offset   a uint64 number of the absolute item number
252      *                     assicated with the tag. Can get from nitems_written.
253      * \param key          the tag key as a PMT symbol
254      * \param value        any PMT holding any value for the given key
255      * \param srcid        optional source ID specifier; defaults to PMT_F
256      */
257     void add_item_tag(unsigned int which_output,
258                       uint64_t abs_offset,
259                       const pmt::pmt_t &key,
260                       const pmt::pmt_t &value,
261                       const pmt::pmt_t &srcid=pmt::PMT_F);
262
263     /*!
264      * \brief Given a [start,end), returns a vector of all tags in the range.
265      *
266      * Range of counts is from start to end-1.
267      *
268      * Tags are tuples of:
269      *      (item count, source id, key, value)
270      *
271      * \param v            a vector reference to return tags into
272      * \param which_input  an integer of which input stream to pull from
273      * \param abs_start    a uint64 count of the start of the range of interest
274      * \param abs_end      a uint64 count of the end of the range of interest
275      */
276     void get_tags_in_range(std::vector<pmt::pmt_t> &v,
277                            unsigned int which_input,
278                            uint64_t abs_start,
279                            uint64_t abs_end);
280
281     /*!
282      * \brief Given a [start,end), returns a vector of all tags in the range
283      * with a given key.
284      *
285      * Range of counts is from start to end-1.
286      *
287      * Tags are tuples of:
288      *      (item count, source id, key, value)
289      *
290      * \param v            a vector reference to return tags into
291      * \param which_input  an integer of which input stream to pull from
292      * \param abs_start    a uint64 count of the start of the range of interest
293      * \param abs_end      a uint64 count of the end of the range of interest
294      * \param key          a PMT symbol key to filter only tags of this key
295      */
296     void get_tags_in_range(std::vector<pmt::pmt_t> &v,
297                            unsigned int which_input,
298                            uint64_t abs_start,
299                            uint64_t abs_end,
```

```
300                             const pmt::pmt_t &key);
301
302   // These are really only for internal use, but leaving them public avoids
303   // having to work up an ever-varying list of friends
304
305  public:
306   gr_block_detail_sptr detail () const { return d_detail; }
307   void set_detail (gr_block_detail_sptr detail) { d_detail = detail; }
308 };
309
310 typedef std::vector<gr_block_sptr> gr_block_vector_t;
311 typedef std::vector<gr_block_sptr>::iterator gr_block_viter_t;
312
313 inline gr_block_sptr cast_to_block_sptr(gr_basic_block_sptr p)
314 {
315   return boost::dynamic_pointer_cast<gr_block, gr_basic_block>(p);
316 }
317
318
319 std::ostream&
320 operator << (std::ostream& os, const gr_block *m);
321
322 #endif /* INCLUDED_GR_BLOCK_H */
```

INTENTIONALLY LEFT BLANK

B. The blockWizard.py Script

```
1  #!/usr/bin/env python
2
3  #This code simplifies the process of writing new blocks. Simply download the
       archive "gr-howto-write-a-block-3.3.0.tar.gz" from
4  #ftp://ftp.gnu.org/gnu/gnuradio/gr-howto-write-a-block-3.3.0.tar.gz , extract
       it to a directory "topdir",
5  #and create your custom block in C++ , placing it in topdir/src/lib . Then
       run this script.
6  #The user only needs to implement the block, for example
7  #particularly "general_work" and "forecast" functions. It handles installation
        of shared library,
8  #and ensures various naming "conventions" (really rigid rules), such as name
       of surrogate friend constructor,
9  #expected by the build system are followed. It handles the proper creation of
       the swig file.
10
11 import os
12 import re
13 import sys
14
15 print "\n!!!"
16 print "This wizard helps build (multiple) signal processing blocks, and places
       them in a single module of a single package"
17 print "You only really need to use this the first time you create a new block;
       then, for subsequent code changes to that same block, just use make and
       sudo make install"
18 print "This script will delete old autotools related files from this directory
       and generate new ones; if you have any concerns, back up before
       proceeding"
19 print "!!!"
20 raw_input("Press enter to continue, Ctrl-C to abort:")
21
22
23 print ("\n Cleaning files from aborted previous runs...")
24 os.system("make clean")
25 os.system("rm -rf src/lib/Makefile.am")
26 os.system("rm -rf src/lib/Makefile")
27 os.system("rm -rf src/lib/Makefile.in")
28 os.system("rm -rf src/lib/.deps")
29 os.system("rm -rf src/lib/*.i")
30 os.system("rm -rf src/lib/Makefile.swig.gen")
31
32 src_headers = list()
33 src_source = list()
34
35 class_inheritance = dict()
36 friend_constructor = dict()
37 constructor = dict() #maps header files to the signature of their constructors
38 destructor = dict() #maps header files to the signature of their destructors
39
```

```
40
41 prefix = raw_input('\n\nSpecify prefix: [prefix]/lib/python[version]/site-
       packages (default = /usr/local): ')
42 if len(prefix) == 0:
43      prefix="/usr/local"
44
45
46 version = raw_input('\n\nSpecify python version: ' + prefix + '/lib/python[
       version]/site-packages (default = 2.6): ')
47 if len(version) == 0:
48      version="2.6"
49
50 install_path = prefix + "/lib/python" + version + "/site-packages"
51 install_path_alt = prefix + "/lib/python" + version + "/dist-packages" #
       installation may occur to here instead
52
53 print('\nThe python command to import this block will be: from [package_name]
       import [module_name]')
54 package_name = raw_input('Enter desired package_name (default=testpackage): ')
55 if len(package_name) == 0:
56      package_name="testpackage"
57
58 module_name = raw_input('Enter desired module_name (default=testmodule): ')
59 if len(module_name) == 0:
60      module_name="testmodule"
61
62 all_files=os.listdir(os.getcwd() + "/src/lib")
63
64 #create lists of the .h files and the .cc files
65 for filename in all_files:
66      #header_pattern = re.compile('[\w]+.h')
67      #suffix_pattern = re.compile('[\w]+.cc')
68
69      if ".h" in filename:
70           src_headers.append(filename)
71
72      if ".cc" in filename:
73           src_source.append(filename)
74
75
76
77 print('\n\nThe python command to create this block will be: object = ' +
       module_name + '.[block_name] ()')
78 i = 0
79 block_names={} #dictionary mapping header file to block name
80 for filename in src_headers:
81      i = i + 1
82      block_name = raw_input('Enter desired block_name corresponding to block
           implemented in ' + filename[:-2] + ' (default=test_block' + str(i) + '
           ,0 for none): ')
83      if len(block_name) == 0:
84           block_name="test_block" + str(i)
85      if block_name == 0:
86           block_name=""
```

```
87      block_names[filename] = block_name
88
89
90
91  #figure out which classes inherit from which blocks (e.g. from gr_block or
        gr_sync_block (allow for helper classes which do not inherit at all)
92  class_dec_pattern = re.compile('[\s]*class[\s\w]+:')
93  for filename in src_headers:
94      f = open(os.getcwd() + "/src/lib/" + filename, 'r')
95      for line in f:
96          if class_dec_pattern.match(line):
97              m = re.search('[\s]+[\w]+$', line) #find the name of the class
                    that is inherited
98              if m:
99                  class_inheritance[filename] = m.group().strip()
100             else:
101                 class_inheritance[filename] = ""
102     f.close()
103
104
105
106 #figure out the signature of the constructor from each .h file
107 # key idea is to look for string of the form          "filename(" (where
        filename is without the .h)
108 for filename in src_headers:
109
110     constructor_string=""
111     constructor_name=filename[:-2] #everything but the .h
112     multi_line=0
113
114     f = open(os.getcwd() + "/src/lib/" + filename, 'r')
115
116     for line in f:
117         if ((constructor_name + "(") in line) or ((constructor_name + "␣(") in
                line): #this line contains a constructor declaration
118             constructor_string=line
119             if ("public" in previous_line) or ("private" in previous_line) or
                    ("protected" in previous_line):
120                 constructor_string = previous_line + constructor_string
121             if ";" in line: #the constructor declaration is all on one line
122                 multi_line=0 #does the constructor declaration span multiple
                        lines?
123                 break
124             else:
125                 multi_line=1
126                 continue
127
128         if multi_line == 1: #the constructor declaration is over multiple
                lines
129             constructor_string = constructor_string + line
130             if ";" in line: #look for end of this constructor declaration, i.e
                    a semicolon
131                 break
132
```

```
133         previous_line = line #save the previous line for future use in case we
                need to look back
134
135     constructor[filename] = constructor_string #there is one constructor per
            class obviously
136
137 f.close()
138
139
140
141
142
143
144
145
146
147 #figure out the signature of the destructor from each .h file
148 # key idea is to look for string of the form          "~filename" (where
        filename is without the .h)
149 for filename in src_headers:
150
151     destructor_string=""
152     destructor_name=filename[:-2] #everything but the .h
153     multi_line=0
154
155     f = open(os.getcwd() + "/src/lib/" + filename, 'r')
156
157     for line in f:
158         if ("~" + destructor_name) in line: #this line contains a destructor
                declaration
159             destructor_string=line
160             if ("public" in previous_line) or ("private" in previous_line) or
                    ("protected" in previous_line):
161                 destructor_string = previous_line + destructor_string
162             if ";" in line: #the destructor declaration is all on one line
163                 multi_line=0 #does the destructor declaration span multiple
                        lines?
164                 break
165             else:
166                 multi_line=1
167                 continue
168
169         if multi_line == 1: #the destructor declaration is over multiple lines
170             destructor_string = destructor_string + line
171             if ";" in line: #look for end of this destructor declaration, i.e
                    a semicolon
172                 break
173
174         previous_line = line #save the previous line for future use in case we
                need to look back
175
176     destructor[filename] = destructor_string #there is one constructor per
            class obviously
177
```

```
178 f.close()
179
180
181
182
183
184
185
186 #figure out the friend function (from .h file) acting as the public interface
        for object construction (allow for helper classes that do not have this)
187 for filename in src_headers:
188
189
190     friend=""
191     target="+_)(*&%%@#NNKSJAHFIUWEROIWEALSKJFD"
192     multi_line=0
193
194     f = open(os.getcwd() + "/src/lib/" + filename, 'r')
195
196     for line in f:
197         if "boost::shared_ptr" in line: #this line contains a typedef, we want
                to know the name of the alias so we can find its declaration
198             target = line.strip().split("␣")[-1][:-1] #get the last word, then
                    drop the semicolon of that last word
199             continue
200
201         if target in line:
202              #friends.append(line.strip())
203              friend=line
204              if ";" in line: #the friend declaration is all on one line
205                 multi_line=0 #does the friend declaration span multiple lines?
                        no
206                 break
207              else:
208                 multi_line=1
209                 continue
210
211         if multi_line == 1: #the friend declaration is over multiple lines
212             #friends.append(line.strip())
213             friend = friend + line
214             if ";" in line: #look for end of this friend declaration, i.e  a
                    semicolon
215                 break
216
217     friend_constructor[filename] = friend
218
219 f.close()
220
221
222
223 print
224 print "
        ********************************************************************************
        "
```

```
225 print "Block will be imported in python as:                from " + package_name
    + " import " + module_name
226
227 for value in block_names.values():
228     if len(value) > 1:
229         print "New objects in python will be made as:           object = " +
              module_name + "." + value + " ()"
230
231
232 print "Detected block header files are:                  ", src_headers
233 print "Detected block source files are:                  ", src_source
234 print "Block classes inherit gnuradio base classes as:   ", class_inheritance
235 print "Class constructors are:                           ", constructor
236 print "Class destructors are:                            ", destructor
237 print "Friend public constructors are:                   ", friend_constructor
238 print "Install path of this package is:                  ", install_path
239 print "
    ********************************************************************************
    "
240
241 raw_input('\nPress enter to continue, or Ctrl C to abort')
242
243
244
245
246 #create .i file
247 swig_i_file = open(os.getcwd() + "/src/lib/" + module_name + ".i", 'w')
248 swig_i_file.write('/* -*- c++ -*- */')
249 swig_i_file.write('\n%include "gnuradio.i"')
250 swig_i_file.write('\n%{')
251 for filename in src_headers:
252     swig_i_file.write('\n#include "' + filename + '"')
253 swig_i_file.write('\n%}')
254
255 for filename in src_headers:
256     if (block_names[filename] != '0'): #check that this really corresponds to
          a block implementation and not helper files
257         swig_i_file.write('\n\nGR_SWIG_BLOCK_MAGIC(' + module_name + ',' +
              block_names[filename] + ');')
258         swig_i_file.write('\n\n' + friend_constructor[filename])
259
260     #write the class definition and constructors/destructors in it
261     swig_i_file.write('\n\n' + 'class ' + filename[:-2] + ' : public ' +
          class_inheritance[filename])
262     swig_i_file.write('\n{')
263     swig_i_file.write('\n\n' + constructor[filename])
264     swig_i_file.write('\n\n' + destructor[filename])
265     swig_i_file.write('\n};')
266
267 swig_i_file.close()
268
269
270 #create Makefile.am file in /src/lib
271 Makefile_am_file = open(os.getcwd() + "/src/lib/Makefile.am", 'w')
```

```
272
273 Makefile_am_file.write('include $(top_srcdir)/Makefile.common')
274 Makefile_am_file.write('\n\ngrinclude_HEADERS = \\')
275 i = 0
276 for filename in src_headers:
277     i = i + 1
278     Makefile_am_file.write('\n\t' + filename)
279     if (i < len(src_headers)):
280         Makefile_am_file.write(' \ ')
281
282 Makefile_am_file.write('\n\n\nTOP_SWIG_IFILES = \\')
283 Makefile_am_file.write('\n\t' + module_name + '.i')
284
285 Makefile_am_file.write('\n\n' + module_name + '_pythondir_category = \\')
286 Makefile_am_file.write('\n\t' + package_name)
287
288 Makefile_am_file.write('\n\n' + module_name + '_la_swig_sources = \\')
289 i = 0
290 for filename in src_source:
291     i = i + 1
292     Makefile_am_file.write('\n\t' + filename)
293     if (i < len(src_source)):
294         Makefile_am_file.write(' \ ')
295
296 Makefile_am_file.write('\n\ninclude $(top_srcdir)/Makefile.swig')
297 Makefile_am_file.write('\n\nBUILT_SOURCES = $(swig_built_sources)')
298 Makefile_am_file.write('\n\nno_dist_files = $(swig_built_sources)')
299
300 Makefile_am_file.close()
301
302 #create Makefile.am in /src/python
303 Makefile_am_file = open(os.getcwd() + "/src/python/Makefile.am", 'w')
304 Makefile_am_file.write('include $(top_srcdir)/Makefile.common')
305 Makefile_am_file.close()
306
307
308 #create Makefile.swig.gen
309 os.system("cp src/lib/Makefile.swig.gen.TEMPLATE src/lib/Makefile.swig.gen")
310 module_command = "sed -i s/testmodule/" + module_name + "/gI src/lib/Makefile.
    swig.gen"
311 package_command = "sed -i s/gnuradio/" + package_name + "/gI src/lib/Makefile.
    swig.gen"
312 os.system(module_command)
313 os.system(package_command)
314
315
316 #The build system expects source files to be of the form   [module_name]_[
    block_name].h    or [module_name]_[block_name].cc
317 #This is based off a gnu radio convention. If our files are NOT in this form,
    copy them over into that form
318 #It also expects the friend public constructor to be of the form [module_name]
    _make_[block_name]
319 #create source files that are named according to gnu radio convention, i.e.
    modulename_blockname.h and .cc (in case they don't exist) so that make is
```

```
    happy
conforming_source_files = list() #list of source files conforming to proper
    gnu radio naming convention
swig_i_file = "src/lib/" + module_name + ".i"
Makefile_am_file = "src/lib/Makefile.am"

for f in block_names.keys(): #loop over all source files
    make_function=""
    ideal_make_function=""

    filename = "src/lib/" + f
    ideal_name = "src/lib/" + module_name + "_" + block_names[f]

    if (not os.path.isfile(ideal_name + ".h")): #the conforming, conventional
        name does not exist; create it
        raw_input("\nSource names do not conform to GNU radio convention!
            Press enter to continue with auto-renaming.....\n")
        tmp_file = ideal_name + ".h"
        cmd = "cp " + filename[:-2] + ".h " + tmp_file
        os.system(cmd)
        conforming_source_files.append(tmp_file)
        rename_command = "sed -i s/" + f[:-2] + "/" + module_name + "_" +
            block_names[f] + "/gI " + tmp_file
        os.system(rename_command)

        #correct the friend public interface name to conform to gnu radio
        if len(friend_constructor[f]) > 0:  #this file has a friend
            constructor
            tokens=friend_constructor[f].split() #splits on any whitespace,
                even consecutive whitespaces which are treated as a single
                whitespace, which we want
            make_function=tokens[1].strip() #the name of the public interface
                friend constructor
            if "();" in make_function:
                make_function=make_function[:-3]
            elif "()" in make_function:
                make_function=make_function[:-2]
            ideal_make_function = module_name + "_make_" + block_names[f]
            rename_command = "sed -i s/" + make_function + "/" +
                ideal_make_function + "/gI " + tmp_file
            os.system(rename_command)

        tmp_file = ideal_name + ".cc"
        cmd = "cp " + filename[:-2] + ".cc " + tmp_file
        os.system(cmd)
        conforming_source_files.append(tmp_file)
        rename_command = "sed -i s/" + f[:-2] + "/" + module_name + "_" +
            block_names[f] + "/gI " + tmp_file
        os.system(rename_command)
        rename_command = "sed -i s/" + make_function + "/" +
            ideal_make_function + "/gI " + tmp_file
        os.system(rename_command)

        #the swig .i file needs to be updated to reflect this name change as
```

```
            well
362         rename_command = "sed -i s/" + f[:-2] + "/" + module_name + "_" +
                block_names[f] + "/gI " + swig_i_file
363         os.system(rename_command)
364         rename_command = "sed -i s/" + make_function + "/" +
                ideal_make_function + "/gI " + swig_i_file
365         os.system(rename_command)
366
367         #the Makefile.am file needs to be updated to reflect this name change
                as well
368         rename_command = "sed -i s/" + f[:-2] + "/" + module_name + "_" +
                block_names[f] + "/gI " + Makefile_am_file
369         os.system(rename_command)
370         rename_command = "sed -i s/" + make_function + "/" +
                ideal_make_function + "/gI " + Makefile_am_file
371         os.system(rename_command)
372
373
374
375 raw_input('Requisite files for autotools have been created. Press enter to
        continue with bootstrap, configure, make, make install; or Ctrl C to abort
        ')
376
377
378
379
380 print ("\n\nRunning bootstrap.....")
381 os.system("./bootstrap")
382
383 print ("\n\nRunning configure.....")
384 os.system("./configure --prefix=" + prefix)
385
386
387 print ("\n\nRunning make.....")
388 os.system("make")
389
390 print ("\n\nRunning sudo make install.....")
391 os.system("sudo make install")
392
393 print ("\n\nMaking " + package_name + " a proper python package.....")
394
395 if os.path.isdir(install_path + "/" + package_name):
396     os.system("sudo touch " + install_path + "/" + package_name + "/__init__.
            py")
397 else:
398     os.system("sudo touch " + install_path_alt + "/" + package_name + "/
            __init__.py")
399
400
401 cleanup = raw_input("Do you want to erase all temporary makefiles and swig
        files? (default=yes)")
402 if ("y" in cleanup) or ("Y" in cleanup) or (len(cleanup) == 0):
403     print ("\n\nDoing final cleanup.....")
404     os.system("make clean")
```

```
405     os.system("rm -rf src/lib/Makefile")
406     os.system("rm -rf src/lib/Makefile.am")
407     os.system("rm -rf src/lib/Makefile.in")
408     os.system("rm -rf src/lib/.deps")
409     os.system("rm -rf src/lib/*.i")
410     os.system("rm -rf src/lib/Makefile.swig.gen")
411
412 if len(conforming_source_files) > 0:
413     cleanup = raw_input("Do you want to erase all renamed source files? (
            default=yes)")
414     if ("y" in cleanup) or ("Y" in cleanup) or (len(cleanup) == 0):
415         #remove re-named source files, they are no longer needed
416         for f in conforming_source_files:
417             cmd = "rm -rf " + f
418             os.system(cmd)
419
420
421
422 print ("\n\nYour block is ready to use")
```

NO. OF COPIES	ORGANIZATION
1 ELEC	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV & ENGRG CTR ARMAMENT ENGRG & TECHNLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD A RIVERA FT HUACHUCA AZ 85613-5300
1	COMMANDER US ARMY RDECOM ATTN AMSRD AMR W C MCCORKLE 5400 FOWLER RD REDSTONE ARSENAL AL 35898-5000
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
8	US ARMY RSRCH LAB ATTN IMNE ALC HRR MAIL & RECORDS MGMT ATTN RDRL CI J PELLEGRINO ATTN RDRL CIN A KOTT ATTN RDRL CIN T B RIVERA ATTN RDRL CIN T G VERMA ATTN RDRL CIN T P YU ATTN RDRL CIO LL TECHL LIB ATTN RDRL CIO MT TECHL PUB ADELPHI MD 20783-1197

INTENTIONALLY LEFT BLANK.