Developing Signal Processing Blocks for Software-defined Radios

by Gunjan Verma and Paul Yu
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.
Developing Signal Processing Blocks for Software-defined Radios

Gunjan Verma and Paul Yu
Computational and Information Sciences Directorate, ARL
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.
## Contents

1. Introduction .................................................. 1

2. Implementation of Blocks ........................................ 2
   2.1 Naming Conventions .......................................... 2
   2.2 Data Types .................................................. 2
      2.2.1 Block Signatures ....................................... 3
      2.2.2 Boost Pointers ......................................... 4
   2.3 Case Study: gr_block ......................................... 4
      2.3.1 Function: general_work() ............................... 6
      2.3.2 Functions: forecast() and set_history() ............. 7
      2.3.3 Field: d_output_multiple ................................ 7
      2.3.4 Field: d_relative_rate .................................. 8

3. Creation of SWIG Interfaces .................................. 8
   3.1 Naming Conventions .......................................... 8
      3.1.1 Block Names ............................................ 9
      3.1.2 Boost Pointers ......................................... 9
   3.2 SWIG Interface File .......................................... 11

4. Installation of Blocks ........................................ 12
   4.1 Directory .................................................... 12
   4.2 Preparing Makefile.am for Autotools ....................... 13
   4.3 Installation ................................................ 14
5. Usage of Blocks

5.1 Invoking from Python

5.2 Debugging

5.3 Simplifying the Build

6. Conclusion

A. The gr_block.h Script

B. The blockWizard.py Script

Distribution
List of Tables

1  Arguments of general_work() ................................. 6
2  Contents of newBlock directory in GNU Radio .................. 13
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 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

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:

```cpp
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

class gr_io_signature { 
  int d_min_streams;
  int d_max_streams;
  std::vector<int> d_sizeof_stream_item;

gr_io_signature(int min_streams, int max_streams,
                 const std::vector<int> &sizeof_stream_items);

friend gr_io_signature_sptr
gr_make_io_signaturev(int min_streams,
                      int max_streams,
                      const std::vector<int> &sizeof_stream_items);

public:
  static const int IO_INFINITE = -1;
  ~gr_io_signature () ;

  int min_streams () const { return d_min_streams; } 
  int max_streams () const { return d_max_streams; } 
  int sizeof_stream_item (int index) const;
  std::vector<int> sizeof_stream_items() const;
};
```

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 following 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

```cpp
std::string d_name;
gr_io_signature_sprr d_input_signature;
gr_io_signature_sprr d_output_signature;
long d_unique_id;
```

### Listing 3. gr_block.h

```cpp
private:
int d_output_multiple;
double d_relative_rate; // approx output_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

```cpp
void set_history (unsigned history) { d_history = history; }
virtual void forecast (int noutput_items,
gr_vector_int &ninput_items_required);
virtual int general_work (int noutput_items,
gr_vector_int &ninput_items,
gr_vector_const_void_star &input_items,
gr_vector_void_star &output_items) = 0;
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>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>noutput_items</td>
<td>Number of output items to write on each output stream</td>
</tr>
<tr>
<td>ninput_items</td>
<td>Number of input items available on each input stream</td>
</tr>
<tr>
<td>input_items</td>
<td>Vector of pointers to elements of the input stream(s), i.e., element $i$ of this vector points to the $i^{th}$ input stream</td>
</tr>
<tr>
<td>output_items</td>
<td>Vector of pointers to elements of the output stream(s), i.e., element $i$ of this vector points to the $i^{th}$ output stream</td>
</tr>
</tbody>
</table>

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 `nininput_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 `nininput_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 is 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:

```c
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:

```c
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:

```python
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

```c++
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:

```cpp
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.

```cpp
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.

```cpp
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`.

```
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)
```

11
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

```python
import gr
```

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

```python
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>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/user/newBlock</td>
<td>Top level Makefile, documentation</td>
</tr>
<tr>
<td>/home/user/newBlock/config</td>
<td>Files for GNU Autotools</td>
</tr>
<tr>
<td>/home/user/newBlock/src</td>
<td>Top level folder for C++ and Python files</td>
</tr>
<tr>
<td>/home/user/newBlock/src/lib</td>
<td>Folder for C++ source/header files</td>
</tr>
</tbody>
</table>

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.
outPython_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 $<
grinclud_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:

```python
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:

```python
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

```c
/* \brief The abstract base class for all 'terminal' processing blocks.

A signal processing flow is constructed by creating a tree of hierarchical blocks, which at any level may also contain terminal nodes that actually implement signal processing functions. This is the base class for all such leaf nodes.

Blocks have a set of input streams and output streams. The input_signature and output_signature define the number of input streams and output streams respectively, and the type of the data items in each stream.

Although blocks may consume data on each input stream at a different rate, all outputs streams must produce data at the same rate. That rate may be different from any of the input rates.

User derived blocks override two methods, forecast and general_work, to implement their signal processing behavior. forecast is called by the system scheduler to determine how many items are required on each input stream in order to produce a given number of output items. */
```


GNU Radio 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.

GNU Radio 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 GNU Radio; see the file COPYING. If not, write to the Free Software Foundation, Inc., 51 Franklin Street, Boston, MA 02110-1234, USA.

#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H

#include <gr_basic_block.h>

#endif
```
* general_work is called to perform the signal processing in the block.
* It reads the input items and writes the output items.
*/

class gr_block : public gr_basic_block {
public:
  //! Magic return values from general_work
  enum {
    WORK_CALLED_PRODUCE = -2,
    WORK_DONE = -1
  };
  
  enum tag_propagation_policy_t {
    TPP_DONT = 0,
    TPP_ALL_TO_ALL = 1,
    TPP_ONE_TO_ONE = 2
  };
  
  virtual ~gr_block () ;

  /!* Assume block computes y_i = f(x_i, x_i-1, x_i-2, x_i-3...)
  * History is the number of x_i's that are examined to produce one y_i.
  * This comes in handy for FIR filters, where we use history to
  * ensure that our input contains the appropriate 'history' for the
  * filter. History should be equal to the number of filter taps.
  */
  unsigned history () const { return d_history ; }
  void set_history (unsigned history) { d_history = history; }

  /!* 
  * \brief Return true if this block has a fixed input to output rate.
  * 
  * If true, then fixed_rate_in_to_out and fixed_rate_out_to_in may be called
  */
  bool fixed_rate () const { return d_fixed_rate ; }

  //
  // override these to define your behavior
  //

  /!* 
  * \brief Estimate input requirements given output request
  * 
  * \param noutput_items number of output items to produce
  * \param ninput_items_required number of input items required on each
  *   input stream
  * 
  * Given a request to produce \p noutput_items, estimate the number of
  * data items required on each input stream. The estimate doesn't have
virtual void forecast(int noutput_items,
                     gr_vector_int &ninput_items_required);

/*! *
 * \brief compute output items from input items
 * *
 * \param noutput_items number of output items to write on each output stream
 * \param ninput_items number of input items available on each input stream
 * \param input_items vector of pointers to the input items, one entry per input stream
 * \param output_items vector of pointers to the output items, one entry per output stream
 * *
 * \returns number of items actually written to each output stream, or \texttt{-1} on EOF.
 * It is OK to return a value less than \texttt{noutput_items}. \texttt{-1} \leq return value \leq \texttt{noutput_items}
 * *
 * general_work must call consume or consume_each to indicate how many items were consumed on each input stream.
 */
virtual int general_work(int noutput_items,
                         gr_vector_int &ninput_items,
                         gr_vector_const_void_star &input_items,
                         gr_vector_void_star &output_items) = 0;

/*! *
 * \brief Called to enable drivers, etc for i/o devices.
 * *
 * This allows a block to enable an associated driver to begin transferring data just before we start to execute the scheduler.
 * The end result is that this reduces latency in the pipeline when dealing with audio devices, usrsps, etc.
 */
virtual bool start();

/*! *
 * \brief Called to disable drivers, etc for i/o devices.
 */
virtual bool stop();

//

/*! *
 * \brief Constrain the noutput_items argument passed to forecast and general_work
 * *
 * set_output_multiple causes the scheduler to ensure that the noutput_items argument passed to forecast and general_work will be an integer multiple
 * of \texttt{\param multiple} The default value of output multiple is \texttt{1}.
 */
void set_output_multiple (int multiple);
int output_multiple () const { return d_output_multiple; }

/*!
 * | brief Tell the scheduler \p how_many_items of input stream \p
 * which_input were consumed.
 * !
 * void consume (int which_input, int how_many_items);
 * !
 * | brief Tell the scheduler \p how_many_items were consumed on each input
 * stream.
 * !
 * void consume_each (int how_many_items);
 * !
 * | brief Tell the scheduler \p how_many_items were produced on output
 * stream \p which_output.
 * *
 * | If the block’s general_work method calls produce, \p general_work must
 * return WORK_CALLED_PRODUCE.
 * !
 * void produce (int which_output, int how_many_items);
 * !
 * | brief Set the approximate output rate / input rate
 * *
 * | Provide a hint to the buffer allocator and scheduler.
 * *
 * | The default relative_rate is 1.0
 * *
 * | decimators have relative_rates < 1.0
 * *
 * | interpolators have relative_rates > 1.0
 * !
 * void set_relative_rate (double relative_rate);
 * !
 * | brief return the approximate output rate / input rate
 * !
 * double relative_rate () const { return d_relative_rate; }
 * !
 * The following two methods provide special case info to the
 * scheduler in the event that a block has a fixed input to output
 * ratio. gr_sync_block, gr_sync_decimator and gr_sync_interpolator
 * override these. If you’re fixed rate, subclass one of those.
 * !
 * !
 * | brief Given ninput samples, return number of output samples that will be
 * produced.
 * *
 * | N.B. this is only defined if fixed_rate returns true.
 * *
 * | Generally speaking, you don’t need to override this.
 * !
 * virtual int fixed_rate_ninput_to_noutput (int ninput);
* | brief Given noutput samples, return number of input samples required to
* N.B. this is only defined if fixed_rate returns true.
* Generally speaking, you don’t need to override this.
*/
virtual int fixed_rate_noutput_to_ninput(int noutput);

/*/  
* | brief Return the number of items read on input stream which_input
*/
uint64_t nitems_read(unsigned int which_input);

/*/  
* | brief Return the number of items written on output stream which_output
*/
uint64_t nitems_written(unsigned int which_output);

/*/  
* | brief Asks for the policy used by the scheduler to moved tags downstream
*/
tag_propagation_policy_t tag_propagation_policy();

/*/  
* | brief Set the policy by the scheduler to determine how tags are moved
downstream.
*/
void set_tag_propagation_policy(tag_propagation_policy_t p);

//

private:

int d_output_multiple;

double d_relative_rate; // approx output_rate / input_rate

g_r_block_detail_sp tr d_detail; // implementation details

unsigned d_history;

bool d_fixed_rate;

tag_propagation_policy_t d_tag_propagation_policy; // policy for moving tags
downstream

protected:

gr_block (const std::string &name,
gr_io_signature_sp tr input_signature,
gr_io_signature_sp tr output_signature);

void set_fixed_rate (bool fixed_rate){ d_fixed_rate = fixed_rate; }
void add_item_tag(unsigned int which_output,
    uint64_t abs_offset, 
    const pmt::pmt_t &key,  
    const pmt::pmt_t &value,  
    const pmt::pmt_t &srcid=pmt::PMT_F);  

void get_tags_in_range(std::vector<pmt::pmt_t> &v, 
    unsigned int which_input,  
    uint64_t abs_start, 
    uint64_t abs_end);  

void get_tags_in_range(std::vector<pmt::pmt_t> &v, 
    unsigned int which_input,  
    uint64_t abs_start, 
    uint64_t abs_end, 
    const pmt::pmt_t &key=pmt::PMT_F);
const pmt::pmt_t &key);

// These are really only for internal use, but leaving them public avoids
// having to work up an ever-varying list of friends

public:
    gr_block_detail_sptr detail () const { return d_detail; }
    void set_detail (gr_block_detail_sptr detail) { d_detail = detail; }
};

typedef std::vector<gr_block_sptr> gr_block_vector_t;
typedef std::vector<gr_block_sptr>::iterator gr_block_viter_t;

inline gr_block_sptr cast_to_block_sptr (gr_basic_block_sptr p)
{
    return boost::dynamic_pointer_cast<gr_block, gr_basic_block>(p);
}

std::ostream&
operator << (std::ostream& os, const gr_block *m);

#endif /* INCLUDED_GR_BLOCK_H */
INTENTIONALLY LEFT BLANK
B. The blockWizard.py Script

```python
#!/usr/bin/env python

# This code simplifies the process of writing new blocks. Simply download the
# archive "gr-howto-write-a-block-3.3.0.tar.gz" from
# ftp://ftp.gnu.org/gnu/gnuradio/gr-howto-write-a-block-3.3.0.tar.gz, extract
# it to a directory 'topdir',
# and create your custom block in C++, placing it in topdir/src/lib. Then
# run this script.
#
The user only needs to implement the block, for example
# particularly 'general_work' and 'forecast' functions. It handles installation
# of shared library,
# and ensures various naming 'conventions' (really rigid rules), such as name
# of surrogate friend constructor,
# expected by the build system are followed. It handles the proper creation of
# the swig file.

import os
import re
import sys

print '"\n!!!"'  # This wizard helps build (multiple) signal processing blocks, and places
# them in a single module of a single package'
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'
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'
print '"!!!"'
raw_input('Press enter to continue, Ctrl-C to abort: ')

print ('
Cleaning files from aborted previous runs...')
os.system('make clean')
os.system('rm -rf src/lib/Makefile.am')
os.system('rm -rf src/lib/Makefile')
os.system('rm -rf src/lib/Makefile.in')
os.system('rm -rf src/lib/*.deps')
os.system('rm -rf src/lib/*.i')
os.system('rm -rf src/lib/Makefile.swig.gen')

src_headers = list()
src_source = list()
class_inheritance = dict()
friend_constructor = dict()
constructor = dict()  # maps header files to the signature of their constructors
destructor = dict()  # maps header files to the signature of their destructors
```
```python
prefix = raw_input('\n\nSpecify \nprefix: [prefix]/lib/python[version]/site-packages (default=/usr/local):\n')
if len(prefix) == 0:
    prefix = '/usr/local'

version = raw_input('\n\nSpecify python version: [prefix]/lib/python[version]/site-packages (default=2.6):\n')
if len(version) == 0:
    version = '2.6'

install_path = prefix + '/lib/python' + version + '/site-packages'
install_path_alt = prefix + '/lib/python' + version + '/dist-packages'  # installation may occur to here instead

print('\n\nThe python command to import this block will be: from [package_name] import [module_name]')
package_name = raw_input('Enter desired package_name (default=testpackage):\n')
if len(package_name) == 0:
    package_name = 'testpackage'

module_name = raw_input('Enter desired module_name (default=testmodule):\n')
if len(module_name) == 0:
    module_name = 'testmodule'

all_files = os.listdir(os.getcwd() + '/src/lib')
# create lists of the .h files and the .cc files
for filename in all_files:
    # header_pattern = re.compile('*[\w]+.h')
    # suffix_pattern = re.compile('*[\w]+.cc')
    if '.h' in filename:
        src_headers.append(filename)
    if '.cc' in filename:
        src_source.append(filename)

print('\n\nThe python command to create this block will be: object = ' + module_name + '.[block_name]()')
i = 0
block_names = {}  # dictionary mapping header file to block name
for filename in src_headers:
    i = i + 1
    block_name = raw_input('Enter desired block name corresponding to block implemented in ' + filename[:-2] + ' (default=test_block_' + str(i) + ', 0 for none):\n')
    if len(block_name) == 0:
        block_name = 'test_block_' + str(i)
    if block_name == 0:
        block_name = ''
```

block_names[filename] = block_name

# 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)
class_dec_pattern = re.compile('^[s]*class[\s]+:')
for filename in src_headers:
    f = open(os.getcwd() + '/src/lib/' + filename, 'r')
    for line in f:
        if class_dec_pattern.match(line):
            m = re.search('[s]+[\w]+$', line) # find the name of the class
            if m:
                class_inheritance[filename] = m.group().strip()
            else:
                class_inheritance[filename] = ''
    f.close()

# figure out the signature of the constructor from each .h file
# key idea is to look for string of the form "filename(" (where
# filename is without the .h)
for filename in src_headers:
    constructor_string=""
    constructor_name=filename[:-2] # everything but the .h
    multi_line=0
    f = open(os.getcwd() + '/src/lib/' + filename, 'r')
    for line in f:
        if ((constructor_name + '(') in line) or ((constructor_name + '\(') in
            line): # this line contains a constructor declaration
            constructor_string=line
            if ('public' in previous_line) or ('private' in previous_line) or
                ('protected' in previous_line):
                constructor_string = previous_line + constructor_string
            if ';.' in line: # the constructor declaration is all on one line
                multi_line=0 # does the constructor declaration span multiple
                lines?
                break
            else:
                multi_line=1
                continue
        if multi_line == 1: # the constructor declaration is over multiple
            lines
            constructor_string = constructor_string + line
            if ';.' in line: # look for end of this constructor declaration, i.e
                a semicolon
                break
previous_line = line  # save the previous line for future use in case we need to look back

constructor[filename] = constructor_string  # there is one constructor per class obviously

f.close()

# figure out the signature of the destructor from each .h file
# key idea is to look for string of the form '~filename' (where filename is without the .h)
for filename in src_headers:
    destructor_string=''    destructor_name=filename[:-2]  # everything but the .h
    multi_line=0
    f = open(os.getcwd() + '/src/lib/' + filename, 'r')

    for line in f:
        if ('~' + destructor_name) in line:  # this line contains a destructor declaration
            destructor_string=line
            if ('public' in previous_line) or ('private' in previous_line) or ('protected' in previous_line):
                destructor_string = previous_line + destructor_string
            if ';' in line:  # the destructor declaration is all on one line
                multi_line=0  # does the destructor declaration span multiple lines?
                break
            else:
                multi_line=1
                continue
        if multi_line == 1:  # the destructor declaration is over multiple lines
            destructor_string = destructor_string + line
            if ';' in line:  # look for end of this destructor declaration, i.e.
                a semicolon
                break

    previous_line = line  # save the previous line for future use in case we need to look back

    destructor[filename] = destructor_string  # there is one constructor per class obviously
for filename in src_headers:

    friend=''
    target='+_(+%%@#NNKSJAHFIUWEROIWEALSJJD*' 
    multi_line=0

    f = open(os.getcwd() + '/src/lib/' + filename, 'r')

    for line in f:
        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
        target = line.strip().split('␣')[-1][-1] #get the last word, then
drop the semicolon of that last word
        continue

        if target in line:
            #friends.append(line.strip())
            friend = line
            if ';;' in line: #the friend declaration is all on one line
                multi_line=0 #does the friend declaration span multiple lines?
                break
            else:
                multi_line=1
                continue

        if multi_line == 1: #the friend declaration is over multiple lines
            #friends.append(line.strip())
            friend += line
            if ';;' in line: #look for end of this friend declaration, i.e a
            semicolon
                break

        friend_constructor[filename] = friend

    f.close()
print 'Block will be imported in python as: ' + package_name + 'import ' + module_name

for value in block_names.values():
    if len(value) > 1:
        print 'New objects in python will be made as: ' + module_name + '. ' + value + '()' 

print 'Detected block header files are: ', src_headers
print 'Detected block source files are: ', src_source
print 'Block classes inherit gnuradio base classes as: ', class_inheritance
print 'Class constructors are: ', constructor
print 'Class destructors are: ', destructor
print 'Friend public constructors are: ', friend_constructor
print 'Install path of this package is: ', install_path

raw_input( 'Press enter to continue, or Ctrl C to abort ')

# create .i file
swig_i_file = open(os.getcwd() + '/src/lib/' + module_name + '.i', 'w')
swig_i_file.write('/∗−∗−c++−∗−/')
swig_i_file.write('
%include "gnuradio.i"
%{

for filename in src_headers:
    swig_i_file.write('n#include "' + filename + '"')
swig_i_file.write('n%

for filename in src_headers:
    if (block_names[filename] != '0'): # check that this really corresponds to a block implementation and not helper files
        swig_i_file.write('n\nGR_SWIG_BLOCK_MAGIC(' + module_name + ', ' + block_names[filename] + ');')
        swig_i_file.write('n\n' + friend_constructor[filename])

# write the class definition and constructors/destructors in it
swig_i_file.write('n\n' + 'class,' + filename + '[:2] + ' + class_inheritance[filename])
swig_i_file.write('n\n' + constructor[filename])
swig_i_file.write('n\n' + destructor[filename])
swig_i_file.write('n\n);')
swig_i_file.close()

# create Makefile.am file in /src/lib
Makefile_am_file = open(os.getcwd() + '/src/lib/Makefile.am', 'w')
Makefile_am_file.write('include $(top_srcdir)/Makefile.common')
Makefile_am_file.write('\n\ninclude_HEADERS = \n
for i in range(len(src_headers)):
    Makefile_am_file.write('
	' + src_headers[i] + '

TOP_SWIG_IFILES = \n
Makefile_am_file.write('

TOP_PYTHONDIR_CATEGORY = \n
for i in range(len(src_source)):
    Makefile_am_file.write('
	' + src_source[i] + '

include $(top_srcdir)/Makefile.swig')

BUILT_SOURCES = $(swig_built_sources)

no_dist_files = $(swig_built_sources)

Makefile_am_file.close()

# create Makefile.am in /src/python
Makefile_am_file = open(os.getcwd() + '/src/python/Makefile.am', 'w')
Makefile_am_file.write('include $(top_srcdir)/Makefile.common')
Makefile_am_file.close()

# create Makefile.swig.gen
os.system('cp src/lib/Makefile.swig.gen.TEMPLATE src/lib/Makefile.swig.gen')
module_command = 'sed -i s/testmodule/' + module_name + '/gI src/lib/Makefile.swig.gen'
package_command = 'sed -i s/gnuradio/' + package_name + '/gI src/lib/Makefile.swig.gen'

os.system(module_command)

os.system(package_command)

# The build system expects source files to be of the form [module_name]/[block_name].h or [module_name]/[block_name].cc
# This is based off a gnuradio convention. If our files are NOT in this form, copy them over into that form
# It also expects the friend public constructor to be of the form [module_name]/_make_[block_name]
# create source files that are named according to gnuradio convention, i.e. modulename_blockname.h and .cc (in case they don't exist) so that make is
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!\nPress 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 + '_' + block_names[f]
        rename_command = 'sed -i s/' + 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 + '/gI' + tmp_file
    os.system(rename_command)

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

rename_command = 'sed -i s/' + f[:2] + '/\' + module_name + '\_' + block_names[f] + '/gI' + swig_i_file
os.system(rename_command)
rename_command = 'sed -i s/' + make_function + '/\' + ideal_make_function + '/gI' + swig_i_file
os.system(rename_command)

# the Makefile.am file needs to be updated to reflect this name change as well
rename_command = 'sed -i s/' + f[:2] + '/\' + module_name + '\_' + block_names[f] + '/gI' + Makefile_am_file
os.system(rename_command)
rename_command = 'sed -i s/' + make_function + '/\' + ideal_make_function + '/gI' + Makefile_am_file
os.system(rename_command)

raw_input('Requisite files for autotools have been created. Press enter to continue with bootstrap, configure, make, make install; or Ctrl-C to abort. ')

print ('

Running bootstrap . . . . .
')

os.system('./bootstrap')

print ('

Running configure . . . . .
')

os.system('./configure --prefix=' + prefix)

print ('

Running make . . . . .
')

os.system('make')

print ('

Running sudo make install . . . . .
')

os.system('sudo_make_install')

print ('

Making ' + package_name + ' a proper python package . . . . .
')

if os.path.isdir(install_path + '/' + package_name):
    os.system('sudo touch ' + install_path + '/' + package_name + '/__init__.py')
else:
    os.system('sudo touch ' + install_path_alt + '/' + package_name + '/__init__.py')

cleanup = raw_input('Do you want to erase all temporary makefiles and swig files? (default=yes) ')

if ('y' in cleanup) or ('Y' in cleanup) or (len(cleanup) == 0):
    print ('

Running final cleanup . . . . .
')

os.system('make clean')
os.system('rm -rf src/lib/Makefile')
os.system('rm -rf src/lib/Makefile.am')
os.system('rm -rf src/lib/Makefile.in')
os.system('rm -rf src/lib/*.dep')
os.system('rm -rf src/lib/*_i')
os.system('rm -rf src/lib/Makefile.swig.gen')

if len(conforming_source_files) > 0:
    cleanup = raw_input("Do you want to erase all renamed source files? (default=yes)")
    if ('y' in cleanup) or ('Y' in cleanup) or (len(cleanup) == 0):
        # remove re-named source files, they are no longer needed
        for f in conforming_source_files:
            cmd = 'rm -rf ' + f
            os.system(cmd)

print ('\n\nYour block is ready to use')
<table>
<thead>
<tr>
<th>NO. OF COPIES</th>
<th>ORGANIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADMNSTR</td>
</tr>
<tr>
<td>ELEC</td>
<td>DEFNS TECHL INFO CTR</td>
</tr>
<tr>
<td></td>
<td>ATTN  DTIC OCP</td>
</tr>
<tr>
<td></td>
<td>8725 JOHN J KINGMAN RD STE 0944</td>
</tr>
<tr>
<td></td>
<td>FT BELVOIR VA 22060-6218</td>
</tr>
<tr>
<td>1</td>
<td>US ARMY RSRCH DEV AND ENGRG CMND</td>
</tr>
<tr>
<td></td>
<td>ARMAMENT RSRCH DEV &amp; ENGRG CMND</td>
</tr>
<tr>
<td></td>
<td>ARMAMENT ENGRG &amp; TECHNLGY CTR</td>
</tr>
<tr>
<td></td>
<td>ATTN  AMSRD AAR AEF T J MATTs</td>
</tr>
<tr>
<td></td>
<td>BLDG 305</td>
</tr>
<tr>
<td></td>
<td>ABERDEEN PROVING GROUND MD 21005-5001</td>
</tr>
<tr>
<td>1</td>
<td>US ARMY INFO SYS ENGRG CMND</td>
</tr>
<tr>
<td></td>
<td>ATTN  AMSSEL IE TD A RIVERA</td>
</tr>
<tr>
<td></td>
<td>FT HUACHUCA AZ 85613-5300</td>
</tr>
<tr>
<td>1</td>
<td>COMMANDER</td>
</tr>
<tr>
<td></td>
<td>US ARMY RDECOM</td>
</tr>
<tr>
<td></td>
<td>ATTN  AMSRD AMR W C MCCORKLE</td>
</tr>
<tr>
<td></td>
<td>5400 FOWLER RD</td>
</tr>
<tr>
<td></td>
<td>REDSTONE ARSENAL AL 35898-5000</td>
</tr>
<tr>
<td>1</td>
<td>US GOVERNMENT PRINT OFF</td>
</tr>
<tr>
<td></td>
<td>DEPOSITORY RECEIVING SECTION</td>
</tr>
<tr>
<td></td>
<td>ATTN  MAIL STOP IDAD J TATE</td>
</tr>
<tr>
<td></td>
<td>732 NORTH CAPITOL ST NW</td>
</tr>
<tr>
<td></td>
<td>WASHINGTON DC 20402</td>
</tr>
<tr>
<td>8</td>
<td>US ARMY RSRCH LAB</td>
</tr>
<tr>
<td></td>
<td>ATTN  IMNE ALC HRR MAIL &amp; RECORDS MGMT</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CI J PELLEGRINO</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIN A KOTT</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIN T B RIVERA</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIN T G VERMA</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIN T P YU</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIO LL TECHL LIB</td>
</tr>
<tr>
<td></td>
<td>ATTN  RDRL CIO MT TECHL PUB</td>
</tr>
<tr>
<td></td>
<td>ADELPHI MD 20783-1197</td>
</tr>
</tbody>
</table>
INTENTIONALLY LEFT BLANK.