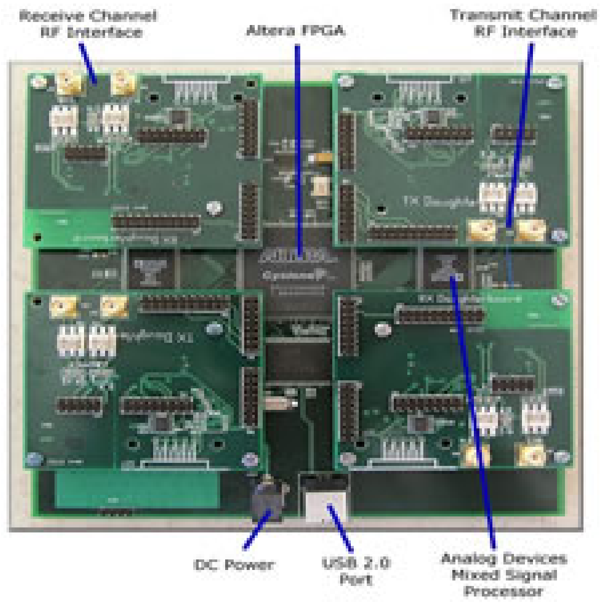
Receive Channel RF Interface
Altera FPGA
Transmit Channel RF Interface
DC Power
USB 2.0 Port
Analog Devices Mixed Signal Processor

GNU Radio ------ TinyOS

Navigation

- Home
- GNU Radio Tutorial
- GNU Radio useful links
- GNU Radio FAQ

Experimental Research in Wirelss Communications and Networks

Currently I am a member of Dr. Laneman's research group, affiliated with the Network Communications & Information Processing (NCIP) Laboratory. My research involves experimental work in wireless communications and networks. Two primary tools are the software radio platform and the sensor motes. The purpose of creating this page, is to provide a room for sharing useful information and resources about these tools and the progress of the projects we are working on.

Why is experiment work important? In my opinion, this is due to the fact that for some problems, especially for some networking problems, there are still no good models yet, and it is difficult to make right assumptions for theoretical work and simulations. So at this stage, experiments may be more crucial.

To play with the sensor motes, the TinyOS programming tool, developed by Berkeley, is essential. The software radio platform we are using now is mainly based on the GNU Radio project. It requires extensive knowledge on programming languages (including C++, python, swig, make etc.), digital signal processing, communications, circuits and hardware design. The project is still at the starting stage, so many open problems are waiting for your effort and contributions.

I am also actively learning these exciting tools. Are you interested? Welcome to send me your comments, questions or discussions.

My email: dshen at nd dot edu

Experimental Research in Wirelss Communications and Networks

Currently I am a member of Dr. Laneman's research group, affiliated with the Network Communications & Information Processing (NCIP) Laboratory. My research involves experimental work in wireless communications and networks. Two primary tools are the software radio platform and the sensor motes. The purpose of creating this page, is to provide a room for sharing useful information and resources about these tools and the progress of the projects we are working on.

Why is experiment work important? In my opinion, this is due to the fact that for some problems, especially for some networking problems, there are still no good models yet, and it is difficult to make right assumptions for theoretical work and simulations. So at this stage, experiments may be more crucial.

To play with the sensor motes, the TinyOS programming tool, developed by Berkeley, is essential. The software radio platform we are using now is mainly based on the GNU Radio project. It requires extensive knowledge on programming languages (including C++, python, swig, make etc.), digital signal processing, communications, circuits and hardware design. The project is still at the starting stage, so many open problems are waiting for your effort and contributions.

I am also actively learning these exciting tools. Are you interested? Welcome to send me your comments, questions or discussions.

My email: dshen at nd dot edu

Dawei's GNU Radio Tutorials

At this stage, GNU Radio project is not well documented. I am writing a set of tutorials to help beginners master this powerful tool. The original purpose of writing these tutorials is to accelerate the learning process for the undergraduate students in Prof. J.N. Laneman's group, who are interested in doing undergraduate reseach. So experienced programmers or engineers may find these tutorials naive.

Welcome to send me your comments or inquiries ^_^. My email is dshen at nd dot edu.

Tutorial 1: GNU Radio Installation Guide - Step by Step

Dawei Shen*

May 19, 2005

Abstract

The tutorials for GNU Radio start from here. The installation procedure is introduced in detail. It is shown that many packages are needed by GNU Radio, which makes the installation bothersome. To avoid trouble, Fedora and Mandrake are recommended. For first time users, it's better to install Linux completely and follow step 1 to step 7 strictly.

The best and easiest way to get GNU Radio software is to build and install it directly from CVS, which is described in Step 8. However, GNU Radio requires **many** other packages pre-installed in your system. Missing any one of them will lead you into trouble. So I suggest for first time installation, it is better to go through 1 until 7 step by step. Later, whenever you want to check new versions of GNU Radio appearing on CVS, you could go to step 8 directly. GNU Radio is designed and tested well in Linux operating system. **Fedora** and **Mandrake** seem to work best at this point. Make sure when you install Linux, the development tools and the x-window tools are included, which can make your life much easier. Complete installation of Linux is a good option for beginners.

1 Downloading necessary packages or tarballs

For our *USRP* users, we need the following modules given in Table 1.

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

Table 1: GNU Radio Modules

gnuradio-core (*)	the 2.x core library
gnuradio-examples	examples for the 2.x tree
gr-audio-oss (*)	soundcard support using OSS
gr-audio-alsa (*)	soundcard support using ALSA
gr-usrp (*)	glue that connects usrp into GNU Radio framework
gr-wxgui (*)	wxPython based GUI tools including FFT and o'scope
gr-howto-write-a-block	examples, Makefiles and article source
gr-gsm-fr-vocoder	GSM 06.10 13kbit/sec vocoder
usrp (*)	USRP board support

The modules marked with '*' are necessary components. These modules can be downloaded directly from the GNU Radio's web page. However, the best way to get the newest version of these modules (except *usrp*) is to checkout and download them from CVS using the following commands:

```
$export CVS_RSH="ssh"
$cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/gnuradio co -P <modulename>
```

The *USRP* CVS tree is hosted on SourceForge. Use these commands to check the usrp module out:

```
$cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/opensdr login
$cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/opensdr co -P usrp
```

Then a directory with the module name will appear in your current folder. Every time you should run `./bootstrap` first to generate the configuration and make files.

2 Installation of gnuradio-core

This is the most important module in GNU Radio, which requires many packages pre-installed in your system. To make our life easier, when we install the GNU/linux in our computer (Fedora 3 for me), we should make sure the **development tools** and **X-window development tools** are checked, so that we can avoid a lot of trouble. It has been determined that GNU Radio exercises bugs in certain versions of g++ 3.3.x on the x86 platform. If you are using g++ 3.3 and make check fails, please either upgrade to 3.4 or downgrade to 3.2. Both are known to work.

Three packages are necessary and often missing:

A. *FFTW*

   Webpage: http://www.fftw.org/

   Download page: http://www.fftw.org/download.html

   IMPORTANT!!! When building *FFTW*, you MUST use the `--enable-single` and `--enable-shared` configure options. You should also use either the `--enable-3dnow` or `--enable-sse` options if you're on an Athlon or Pentium respectively. Note that sometimes you may meet the 'segmentation fault' error when you try to run a python code. You can remove the `--enable-sse` option to avoid this problem.

B. *cppunit*

   Webpage: http://cppunit.sourceforge.net

   Download page: https://sourceforge.net/project/showfiles.php?group_id=11795

C. *SWIG*

   Webpage: http://www.swig.org

   Download page: http://sourceforge.net/projects/swig/

Two Python modules are necessary and very important in GNU Radio, *Numeric* and *Numarray*, which are used in Python for numerical computation purpose. They are available at:

http://sourceforge.net/projects/numpy

Download the newest version of *Numarray* and *Numpy*. Extract the tarball then you can find the **setup.py** file. Use the following command to install

```
$python setup.py install
```

At this point, I guess you should be able to install *gnuradio-core* without losing any packages. In the folder *gnuradio-core* downloaded from CVS, run the following commands

```
$ ./bootstrap
$ ./configure --enable-maintainer-mode
$ make
$ make check
$ make install
```

Now you have already got the key component of GNU Radio. Note that you may not be able to pass the 'make check' if you didn't install *Numeric* and *Numarray* modules.

3 Installation of gr-audio-oss, gr-audio-alsa

Nothing particular is worth mentioning here, they are two modules providing you the access to your sound card. Remember to run `./bootstrap` first to generate the configuration and make files.

4 Installation of gr-wxgui

gr-wxgui provides the graphical FFT and o'scope in GNU Radio and is based on *wxPython*. So certainly we should install *wxPython* first.

Webpage: http://www.wxPython.org/
Download page: https://sourceforge.net/project/showfiles.php?group_id=10718

There are two types of *wxPython*, which are ansi based and unicode based. Either one of them works. Pay attention to the suffix and choose the correct rpms for your Python (2.3 or 2.4). For example, in my computer, I choose the following three rpms.

```
wxPython2.5-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
wxPython-common-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
wxPython2.5-devel-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
```

Download them into one folder and simply run

```
$ rpm -ivh *.rpm
```

Now *wxPython* has been installed and you can install *gr-wxgui* without any problem.

5 Installation of usrp and gr-usrp

The *usrp* module provides necessary drivers, fpga controls, firmware interfaces for GNU Radio's use. It must be installed before *gr-usrp*. We'll need the *SDCC* free C compiler to build the firmware. *SDCC* can be found at

Webpage: http://sdcc.sourceforge.net/

Download page: http://sourceforge.net/project/showfiles.php?group_id=599

The version 2.4 is necessary to avoid some problems. Now you can install *usrp* and *gr-usrp* in order.

6 Installation of gnuradio-examples, gr-howto-write-a-block and gr-gsm-fr-vocoder

Actually these modules are not parts of the GNU Radio software. They just provide some examples, applications or technical articles. They can be installed easily. Note that it's not necessary to 'install' *gnuradio-examples*, you can find many good examples in the `/python` folder.

7 Post-installation tasks

After installation, all the components mentioned above are installed in

```
/usr/local/lib/python2.3/site-packages
```

If you are using python 2.4, then the path should be

```
/usr/local/lib/python2.4/site-packages
```

which is not included in Python's working path by default. To let the Python find these modules and for convenience, you'll need to set the *PYTHONPATH* environment variable as follows:

```
$ export PYTHONPATH=/usr/local/lib/python2.3/site-packages
```

You may want to add this to your `~/.bash_profile`.

Two important documents are located at

```
/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html
/usr/local/share/doc/usrp-x.xcvs/html/index.html
```

They are the documentations for *gnuradio* and *usrp*. You may like to bookmark them in your web browser.

8 How to check the updates and install the updated versions of GNU Radio software from CVS?

Note that the steps mentioned above are only for first-time installation, because typically quite a lot of packages are missing at this point. Once you have installed GNU Radio successfully, you have a much easier way to check the updates and install the updated versions of GNU Radio from CVS. Here's the step-by-step version.

We checkout *gr-build* by hand, then use it to bootstrap everything else.

```
$ export CVS_RSH="ssh"

$ cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/gnuradio co -P gr-build
```

You now have a directory called *gr-build.* Change directory into it:

```
$cd gr-build
```

This checkout command will checkout everything except for that related to the Measurement Computing PCI card.

```
$./checkout -x mc4020
```

You'll now have a bunch of directories...

If you haven't already, you'll want to set up **/etc/sudoers**. The buildit script below does the bulk of the work running as you, then uses '**sudo make install**' to install the stuff into **/usr/local/**...

First, add yourself to group **wheel** by editing **/etc/group**. Find the line that starts with **wheel:x:**... and add your log name to the end of it.

Then make these changes to **/etc/sudoers**:

```
# Defaults specification
# Only ask for password every 10 minutes.
Defaults timestamp_timeout=10
# Uncomment to allow people in group wheel to run all commands
%wheel ALL=(ALL) ALL
```

Now, assuming you've got all the dependencies fulfilled, this will bootstrap, configure, make, make check, and make install everything in the proper order:

```
$sudo -v # give sudo the password now, so you can walk away while it builds
```

Now build everything that you checked out

```
$./for-all-dirs ../buildit 2>&1 | tee make.log
```

Assuming the build completed successfully, you're all set.

Later when you want to see if there are updates to CVS head, but don't want to apply them to your tree use this:

```
$./for-all-dirs cvs -nqP up
```

If you like what you saw from the previous command, this will get the updates and merge them into your tree:

```
$./for-all-dirs cvs -qP up
```

Then you should rebuild everything:

```
$./for-all-dirs ../buildit
```

References

[1] GNU Radio on-line document: **How to Build from CVS**,
http://comsec.com/wiki?HowtoBuildFromCVS

Tutorial 1: GNU Radio Installation Guide - Step by Step

Dawei Shen[1]

May 19, 2005

Abstract

The tutorials for GNU Radio start from here. The installation procedure is introduced in detail. It is shown that many packages are needed by GNU Radio, which makes the installation bothersome. To avoid trouble, Fedora and Mandrake are recommended. For first time users, it's better to install Linux completely and follow step 1 to step 7 strictly.

Contents

The best and easiest way to get GNU Radio software is to build and install it directly from CVS, which is described in Step 8. However, GNU Radio requires **many** other packages pre-installed in your system. Missing any one of them will lead you into trouble. So I suggest for first time installation, it is better to go through 1 until 7 step by step. Later, whenever you want to check new versions of GNU Radio appearing on CVS, you could go to step 8 directly. GNU Radio is designed and tested well in Linux operating system. **Fedora** and **Mandrake** seem to work best at this point. Make sure when you install Linux, the development tools and the x-window tools are included, which can make your life much easier. Complete installation of Linux is a good option for beginners.

1 Downloading necessary packages or tarballs

For our *USRP* users, we need the following modules given in Table 1.

Table 1: GNU Radio Modules

gnuradio-core (*)	the 2.x core library
gnuradio-examples	examples for the 2.x tree
gr-audio-oss (*)	soundcard support using OSS

gr-audio-alsa (*)	soundcard support using ALSA
gr-usrp (*)	glue that connects usrp into GNU Radio framework
gr-wxgui (*)	wxPython based GUI tools including FFT and o'scope
gr-howto-write-a-block	examples, Makefiles and article source
gr-gsm-fr-vocoder	GSM 06.10 13kbit/sec vocoder
usrp (*)	USRP board support

The modules marked with `*' are necessary components. These modules can be downloaded directly from the GNU Radio's web page. However, the best way to get the newest version of these modules (except *usrp*) is to checkout and download them from CVS using the following commands:

```
$export CVS_RSH="ssh"
$cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/gnuradio co -
P <modulename>
```

The *USRP* CVS tree is hosted on SourceForge. Use these commands to check the usrp module out:

```
$cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/
opensdr login
$cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/
opensdr co -P usrp
```

Then a directory with the module name will appear in your current folder. Every time you should run `./bootstrap` first to generate the configuration and make files.

2 Installation of gnuradio-core

This is the most important module in GNU Radio, which requires many packages pre-installed in your system. To make our life easier, when we install the GNU/linux in our computer (Fedora 3 for me), we should make sure the **development tools** and **X-window development tools** are checked, so that we can avoid a lot of trouble. It has been determined that GNU Radio exercises bugs in certain versions of g++ 3.3.x on the x86 platform. If you are using g++ 3.3 and make check fails, please either upgrade to 3.4 or downgrade to 3.2. Both are known to work.

Three packages are necessary and often missing:

A. FFTW

Webpage:http://www.fftw.org/
Download page: http://www.fftw.org/download.html

IMPORTANT!!! When building *FFTW*, you MUST use the `--enable-single` and `--enable-shared` configure options. You should also use either the `--enable-3dnow` or `--enable-sse` options if you're on an Athlon or Pentium respectively. Note that sometimes you may meet the `segmentation fault' error when you try to run a python code. You can remove the `--enable-sse` option to avoid this problem.

B. cppunit

Webpage:http://cppunit.sourceforge.net
Download page: https://sourceforge.net/project/showfiles.php?group_id=11795

C. SWIG

Webpage:http://www.swig.org
Download page:http://sourceforge.net/projects/swig/

Two Python modules are necessary and very important in GNU Radio, *Numeric* and *Numarray*, which are used in Python for numerical computation purpose. They are available at:

http://sourceforge.net/projects/numpy

Download the newest version of *Numarray* and *Numpy*. Extract the tarball then you can find the **setup.py** file. Use the following command to install

```
$python setup.py install
```

At this point, I guess you should be able to install *gnuradio-core* without losing any packages. In the folder *gnuradio-core* downloaded from CVS, run the following commands

```
$ ./bootstrap
$ ./configure --enable-maintainer-mode
$ make
$ make check
$ make install
```

Now you have already got the key component of GNU Radio. Note that you may not be able to pass the `make check' if you didn't install *Numeric* and *Numarray* modules.

3 Installation of gr-audio-oss, gr-audio-alsa

Nothing particular is worth mentioning here, they are two modules providing you the access to your sound card. Remember to run `./bootstrap` first to generate the configuration and make files.

4 Installation of gr-wxgui

gr-wxgui provides the graphical FFT and o'scope in GNU Radio and is based on *wxPython*. So certainly we should install *wxPython* first.

Webpage:
http://www.wxPython.org/
Download page:
https://sourceforge.net/project/showfiles.php?group_id=10718

There are two types of *wxPython*, which are ansi based and unicode based. Either one of them works. Pay attention to the suffix and choose the correct rpms for your Python (2.3 or 2.4). For example, in my computer, I choose the following three rpms.

```
wxPython2.5-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
wxPython-common-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
wxPython2.5-devel-gtk2-unicode-2.5.4.1-fc2_py2.3.i386.rpm
```

Download them into one folder and simply run

```
$ rpm -ivh *.rpm
```

Now *wxPython* has been installed and you can install *gr-wxgui* without any problem.

5 Installation of usrp and gr-usrp

The *usrp* module provides necessary drivers, fpga controls, firmware interfaces for GNU Radio's use. It must be installed before *gr-usrp*. We'll need the *SDCC* free C compiler to build the firmware. *SDCC* can be found at

Webpage:
http://sdcc.sourceforge.net/
Download page:
http://sourceforge.net/project/showfiles.php?group_id=599

The version 2.4 is necessary to avoid some problems. Now you can install *usrp* and *gr-usrp* in order.

6 Installation of gnuradio-examples, gr-howto-write-a-block and gr-gsm-fr-vocoder

Actually these modules are not parts of the GNU Radio software. They just provide some examples, applications or technical articles. They can be installed easily. Note that it's not necessary to `install' *gnuradio-examples*, you can find many good examples in the `/python` folder.

7 Post-installation tasks

After installation, all the components mentioned above are installed in

```
    /usr/local/lib/python2.3/site-packages
```

If you are using python 2.4, then the path should be

```
    /usr/local/lib/python2.4/site-packages
```

which is not included in Python's working path by default. To let the Python find these modules and for convenience, you'll need to set the *PYTHONPATH* environment variable as follows:

```
    $ export PYTHONPATH=/usr/local/lib/python2.3/site-packages
```

You may want to add this to your `~/.bash_profile`.

Two important documents are located at

```
    /usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html
    /usr/local/share/doc/usrp-x.xcvs/html/index.html
```

They are the documentations for *gnuradio* and *usrp*. You may like to bookmark them in your web browser.

8 How to check the updates and install the updated versions of GNU Radio software from CVS?

Note that the steps mentioned above are only for first-time installation, because typically quite a lot of packages are missing at this point. Once you have installed GNU Radio successfully, you have a much easier way to check the updates and install the updated versions of GNU Radio from CVS. Here's the step-by-step version.

We checkout *gr-build* by hand, then use it to bootstrap everything else.

```
    $ export CVS_RSH="ssh"
    $ cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/gnuradio co -
    P gr-build
```

You now have a directory called *gr-build*. Change directory into it:

```
    $cd gr-build
```

This checkout command will checkout everything except for that related to the Measurement Computing PCI card.

```
    $./checkout -x mc4020
```

You'll now have a bunch of directories...

If you haven't already, you'll want to set up **/etc/sudoers**. The buildit script below does the bulk of the work running as you, then uses `**sudo make install**' to install the stuff into **/usr/local/**...

First, add yourself to group **wheel** by editing **/etc/group**. Find the line that starts with **wheel:x:**... and add your log name to the end of it.

Then make these changes to **/etc/sudoers**:

```
# Defaults specification
# Only ask for password every 10 minutes.
Defaults timestamp_timeout=10
# Uncomment to allow people in group wheel to run all commands
%wheel ALL=(ALL) ALL
```

Now, assuming you've got all the dependencies fulfilled, this will bootstrap, configure, make, make check, and make install everything in the proper order:

```
$sudo -
v # give sudo the password now, so you can walk away while it builds
```

Now build everything that you checked out

```
$./for-all-dirs ../buildit 2>&1 | tee make.log
```

Assuming the build completed successfully, you're all set.

Later when you want to see if there are updates to CVS head, but don't want to apply them to your tree use this:

```
$./for-all-dirs cvs -nqP up
```

If you like what you saw from the previous command, this will get the updates and merge them into your tree:

```
$./for-all-dirs cvs -qP up
```

Then you should rebuild everything:

```
$./for-all-dirs ../buildit
```

References

[1]

GNU Radio on-line document: **How to Build from CVS**, http://comsec.com/wiki?HowtoBuildFromCVS

Footnotes:

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

---

Tutorial 2: Before Diving into GNU Radio, You Should ...

Dawei Shen[*]

May 21, 2005

Abstract

GNU Radio requires both strong computer skills and extensive knowledge on communications and digital signal processing. This article lists some useful resources including textbooks, web links, on-line tutorials. The purpose of this sheet is to help the GNU Radio fans to get prepared for this exciting tool.

I definitely believe you have found GNU Radio interesting and are eager to play around with it. Unfortunately, it contains more challenge than fun. You need extensive knowledge on a wide range of areas, including communication (wireless) systems, digital signal processing, basic hardware and circuit design, OOP programming, etc. However, your interest and passion could make them much easier. In this page, I list some useful articles and resources, which might be important to you before you dive into the GNU Radio.

1 Having a clearer picture of GNU Radio ...

If you haven't, please read this on-line article first. *Eric Blossom*, "*Exploring GNU Radio*". A very brief and excellent introduction to software radio. Eric is the founder of the whole GNU Radio project. Make sure you understand how ADC works and why we need an RF frontend. Recall the sampling theory learnt from your Signal and Systems class. Then take a close look at the two parts "**The Universal Software Radio Peripheral**" and "**What Goes in the FPGA**". This article also provides two examples, a simple dial tone output and an FM receiver. At least you should understand the first one. Can't understand the FM receiver? Never mind, read the second article: *Eric Blossom*, "*Listen to FM Radio in Software, Step by Step*". You don't need to understand the code line by line, but you should know how the signal flows from the air to the soundcard. Then it's better for you to know more about what USRP does, the two pages "*USRP wiki*" and "*USRP User's Guide*" would be very valuable. Assuming you have gone thourgh the articles above, you may go to the "*GNU Radio wiki*" pages to look for more information.

2 Programming on GNU Radio ...

To really "play" with GNU Radio, you should be able to write your own code. From the article "Exploring GNU Radio", you should have known that the software structure of GNU Radio contains two levels. All the signal processing blocks are written in C++ and Python is used to create a network or graph and glue these blocks together. So in this particular scenario, Python is a higher level language. Many useful and frequenctly used blocks have been provided by the GNU Radio project, so in many cases you don't need to touch C++, just using Python to finish your task. However, to do more sophisticated work, you have to use C++ to create your own block. In such cases, an on-line

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

article *Eric Blossom*, "*how to write a block*" is what you need. You may want to know, what blocks have been provided to us? Unfortunately, unlike some other developing tools, such as TinyOS, GNU Radio is badly documented at this point. But you still have two very useful documents generated using Doxygen. After installing the "*gnuradio-core*" and "*usrp*" modules, you can find two html packages located at

- /usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html
- /usr/local/share/doc/usrp-x.xcvs/html/index.html

I bookmark them in my brower. Although they are not clear enough, they can tell you quite much information. The first one is also available on-line here.

If you haven't had a chance to use Python, please go though the Python on-line tutorials. The most important sections are:

- Section 2: Using the Python Interpreter
- Section 3 An Informal Introduction to Python
- Section 6: Modules
- Section 7: Input and Output
- Section 9: Classes

They will be frequently used in GNU Radio programming. If Object-Oriented Programming (OOP) sounds unfamiliar to you, you should read section 9 more carefully. The following links also may help you to grasp the essence of OOP.

- Lesson: Object-Oriented Programming Concepts
- Introduction to Object-Oriented Programming Using C++
- The Object Oriented Programming Web

Anyway, Python seems to be crucial at this stage, so make sure you master it well.

3 Digital Signal Processing (DSP)

Most of us have taken the "Signals and Systems" class, I guess. What we learnt in this course is extremely important here. However, that's not enough. Make sure you won't get lost if I change the signal to analog or to digital world; to the time domain or to the frequency domain. The bottom line is, you need to know what is sampling theorem, what is z-transform, how to get the spectrum of a signal, and the concepts of FIR and IIR filters. I recommend several books here, which are famous and classical.

- "**Signals and Systems (2nd edition)**" - Alan V. Oppenheim, Alan S. Willsky
- "**Discrete-Time Singal Processing (2nd edition)**" - Alan V. Oppenheim, Ronald W. Schafer, John R. Buck
- "**Digital Signal Processing: Principles, Algorithms and Applications (3rd edition)**" - John G. Proakis, Dimitris Manolakis

Read the chapters about discrete-time Fourier tranform and FIR, IIR filters. I know the books are expensive and tedious, there are some other useful on-line resources:

- Digital Signal Processing Tutorial
- The Scientist and Engineer's Guide to Digital Signal Processing

4 Communications

We know the real signals we send and receive can't be in base band. They need to modulated and demodulated. I believe you have studied the concepts of AM and FM radios in some courses. Both of them belong to analog world. To develop more useful and interesting schemes, we need digital communications. Of particular importance and interest at this point, is digital modulation and demodulation, synchronization. The course in your senior year, "Communication Systems" may be of interest to you. Further more, I recommend you read:

- Chapters 4 and 5, "**Digital Communications (4th edition)**" - John G. Proakis

The knowledge introduced in these two chapters is exactly what we are going to try out recently.

5 Ready to start?

I list four topics above. It doesn't mean you should complete them one by one before you can do something with GNU Radio. You certainly can learn them through this study process. But at least you should read the articles I mentioned above in section 1 and section 2. Then you can try this "homework":

In the module "*gnuradio-examples*", you can find many sample codes in the folder *gnuradio-example/python/usrp/*. Can you read the following two programs and understand the code line by line?

- gnuradio-examples/python/usrp/am_rcv.py
- gnuradio-examples/python/usrp/wfm_rcv_gui.py

If so, I would say you have moved a huge step.

References

All the book chapters, on-line resources mentioned in this article.

Tutorial 2: Before Diving into GNU Radio, You Should ...

Dawei Shen[1]

May 21, 2005

Abstract

GNU Radio requires both strong computer skills and extensive knowledge on communications and digital signal processing. This article lists some useful resources including textbooks, web links, on-line tutorials. The purpose of this sheet is to help the GNU Radio fans to get prepared for this exciting tool.

Contents

I definitely believe you have found GNU Radio interesting and are eager to play around with it. Unfortunately, it contains more challenge than fun. You need extensive knowledge on a wide range of areas, including communication (wireless) systems, digital signal processing, basic hardware and circuit design, OOP programming, etc. However, your interest and passion could make them much easier. In this page, I list some useful articles and resources, which might be important to you before you dive into the GNU Radio. There is also a very good reading list suggested by the GNU Radio community, where you can find more useful information.

1 Having a clearer picture of GNU Radio ...

If you haven't, please read this on-line article first. *Eric Blossom*, `*Exploring GNU Radio*'. A very brief and excellent introduction to software radio. Eric is the founder of the whole GNU Radio project. Make sure you understand how ADC works and why we need an RF frontend. Recall the sampling theory learnt from your Signal and Systems class. Then take a close look at the two parts `**The Universal Software Radio Peripheral**' and `**What Goes in the FPGA**'.

This article also provides two examples, a simple dial tone output and an FM receiver. At least you should understand the first one. Can't understand the FM receiver? Never mind, read the second article: *Eric Blossom*, `*Listen to FM Radio in Software, Step by Step*'. You don't need to understand the code line by line, but you should know how the signal flows from the air to the sound card. Then it's better for you to know more about what USRP does, the two pages `*USRP wiki*' and `*USRP User's Guide*' would be very valuable. Assuming you have gone through the articles above, you may go to the `*GNU Radio wiki*' pages to look for more information.

2 Programming on GNU Radio ...

To really `play' with GNU Radio, you should be able to write your own code. From the article `Exploring GNU Radio', you should have known that the software structure of GNU Radio contains two levels. All the signal processing blocks are written in C++ and Python is used to create a network or graph and glue these blocks together. So in this particular scenario, Python is a higher level language. Many useful and frequently used blocks have been provided by the GNU Radio project, so in many cases you don't need to touch C++, just using Python to finish your task. However, to do more sophisticated work, you have to use C++ to create your own block. In such cases, an on-line article *Eric Blossom*, `*how to write a block*' is what you need. You may want to know, what blocks have been provided to us? Unfortunately, unlike some other developing tools, such as TinyOS, GNU Radio is badly documented at this point. But you still have two very useful documents generated using Doxygen. After installing the `*gnuradio-core*' and `*usrp*' modules, you can find two html packages located at

```
/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.
html
/usr/local/share/doc/usrp-x.xcvs/html/index.html
```

I bookmark them in my browser. Although they are not clear enough, they can tell you quite much information. The first one is also available on-line here.

If you haven't had a chance to use Python, please go though the Python on-line tutorials. The most important sections are:

- Section 2: Using the Python Interpreter
- Section 3: An Informal Introduction to Python
- Section 6: Modules
- Section 7: Input and Output
- Section 9: Classes

They will be frequently used in GNU Radio programming. If Object-Oriented Programming (OOP) sounds unfamiliar to you, you should read section 9 more carefully. The following links also may help you to grasp the essence of OOP.

- Lesson: Object-Oriented Programming Concepts
- Introduction to Object-Oriented programming Using C++
- The Object Oriented Programming Web

Anyway, Python seems to be crucial at this stage, so make sure you master it well.

3 Digital Signal Processing (DSP)

Most of us have taken the `Signals and Systems' class, I guess. What we learnt in this course is extremely important here. However, that's not enough. Make sure you won't get lost if I change the signal to analog or to digital world; to the time domain or to the frequency domain. The bottom line is, you need to know what is sampling theorem, what is z-transform, how to get the spectrum of a signal, and the concepts of FIR and IIR filters. I recommend several books here, which are famous and classical.

- **`Signals and Systems (2nd edition)**' - Alan V. Oppenheim, Alan S. Willsky
- **`Discrete-Time Signal Processing (2nd edition)**' - Alan V. Oppenheim, Ronald W. Schafer, John R. Buck
- **`Digital Signal Processing: Principles, Algorithms and Applications (3rd edition)**' - John G. Proakis, Dimitris Manolakis

Read the chapters about discrete-time Fourier transform and FIR, IIR filters. I know the books are expensive and tedious, there are some other useful on-line resources:

- Digital Signal Processing Tutorial
- The Scientist and Engineer's Guide to Digital Signal Processing

4 Communications

We know the real signals we send and receive can't be in base band. They need to modulated and demodulated. I believe you have studied the concepts of AM and FM radios in some courses. Both of them belong to analog world. To develop more useful and interesting schemes, we need digital communications. Of particular importance and interest at this point, is digital modulation and demodulation, synchronization. The course in your senior year, `Communication Systems' may be of interest to you. Further more, I recommend you read:

- Chapters 4 and 5, `**Digital Communications (4th edition)**' - John G. Proakis

The knowledge introduced in these two chapters is exactly what we are going to try out recently.

Here is an awesome book:

- `**Digital Signal Processing in Communication Systems**' - Marvin E. Frerking

This book is practical engineering focus and it contains lots of great examples. Frerking often provides multiple solutions for a given transmitter or receiver design problem. Practical algorithms are proposed rather than purely theoretical discussion. It can be used as a `dictionary' for communication system design.

5 Ready to start?

I list four topics above. It doesn't mean you should complete them one by one before you can do something with GNU Radio. You certainly can learn them through this study process. But at least you should read the articles I mentioned above in section 1 and section 2. Then you can try this `homework':

In the module `*gnuradio-examples*', you can find many sample codes in the folder *gnuradio-example/python/usrp/*. Can you read the following two programs and understand the code line by line?

- `gnuradio-examples/python/usrp/am_rcv.py`
- `gnuradio-examples/python/usrp/wfm_rcv_gui.py`

If so, I would say you have moved a huge step.

References

All the book chapters, on-line resources mentioned in this article.

Footnotes:

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

---

Tutorial 3: Entering the World of GNU Software Radio

Dawei Shen[*]

August 3, 2005

Abstract

This article provides an overview of the GNU Radio toolkit for building software radios. This tutorial is a modified version of Eric's article '*Exploring GNU Radio*'. The concepts of software defined radio and the basics of signal processing will be introduced. We are going to enter the exciting world of GNU Software Radio!

1 Introduction to GNU Software Radio

Software radio is the technique of getting the code as close to the antenna as possible. It turns radio hardware problems into software problems. The fundamental characteristic of software radio is that software defines the transmitted waveforms, and software demodulates the received waveforms. This is in contrast to most radios in which the processing is done with either analog circuitry or analog circuitry combined with digital chips.

Software radio is a revolution in radio design due to its ability to create radios that change on the fly, creating new choices for users. At the baseline, software radios can do pretty much anything a traditional radio can do. The exciting part is the flexibility that software provides us. Controlling a computer, with necessary hardware supports, to play with radios should be easier, interesting and attractive.

GNU Radio is a free software toolkit for learning about, building and deploying software radios. GNU Radio provides a library of signal processing blocks and the glue to tie it all together. It is free and open source, it comes with complete source code so anyone can look and see how the system is built.

2 The structure of a software radio system

2.1 Block diagram

This is a typical RX path for a software radio:

```
Antenna -> Receive RF Front End -> ADC -> Software Code
```

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

This is a typical TX path for a software radio:

```
Software Code -> DAC -> Transmit RF Front End -> Antenna
```

To understand the software part of the radio, we first need to understand a bit about the associated hardware. Examining the receive path in the figure, we see an antenna, a mysterious RF front end, an analog-to-digital converter (ADC) and a bunch of code.

2.2 Analog to Digital Converter (ADC)

The analog-to-digital converter is the bridge between the physical world of continuous analog signals and the world of discrete digital samples manipulated by software.

ADCs have two primary characteristics, sampling rate and dynamic range. Sampling rate is the number of times per second that the ADC measures the analog signal. Dynamic range refers to the difference between the smallest and largest signal that can be distinguished; it's a function of the number of bits in the ADC's digital output and the design of the converter. For example, an 8-bit converter at most can represent 256 (28) signal levels, while a 16-bit converter represents up to 65,536 levels. Generally speaking, device physics and cost impose trade-offs between the sample rate and dynamic range.

After getting through the ADC, the continuous signal becomes a sequence of numbers entering the computer, which can be processed digitally as an array in the software. So what kind of ADC could be a good choice if we want to play with GNU Radio? The best answer should be the USRP board, which is developed by Matt and Eric. We will talk about the USRP board later.

2.3 The RF Front End

To understand the role of the RF front end, we need to talk about a bit of theory. Nyquist theorem tells us that, to avoid aliasing when converting from analog to digital, the ADC sampling frequency must be at least twice the maximum frequency of the signal of interest, in order to sustain all the spectrum information accurately. Aliasing is what makes the wagon wheels look like they're going backward in the old westerns: the sampling rate of the movie camera is not fast enough to represent the position of the spokes unambiguously.

Assuming we're dealing with low pass signals - signals where the bandwidth of interest goes from 0 to f_{MAX}, the Nyquist criterion states that our sampling frequency needs to be at least $2 * f_{MAX}$. But if our ADC runs at 20 MHz, how can we listen to broadcast FM radio at 92.1 MHz? The answer is the RF front end. The receive RF front end translates a range of frequencies appearing at its input (RF band) to a lower range at its output (IF band). For example, we could imagine an RF front end that translated the signals occurring in the 90 - 100 MHz range (RF) down to the 0 - 10 MHz range (IF).

Mostly, we can treat the RF front end as a black box with a single control, the center of the input range that's to be translated. As a concrete example, a cable modem tuner module translates a 6 MHz chunk of the spectrum centered between about 50 MHz and 800 MHz down to an output range centered at 5.75 MHz. The center frequency of the output range is called the intermediate frequency, or IF.

What can be used as the RF front end? If you like to build everything from scratch, you could purchase the Minicircuits parts for prototype. A typical structure is shown below:

```
Antenna -> Low Noise Amplifer (LNA) -> Low Pass Filter (LPF) -> Mixer -> LPF ->ADC
                                           Local Oscillator -> -^
```

The low noise amplifier (LNA) and the low pass filter (LPF) are used to select the bandwidth of interest and amplify the signal. For example, in order to receive the FM stations, you may like to use an LNA and an LPF with a cutoff frequency of 120MHz. A mixer can be treated as a multiplier, with two inputs (the RF signal and the local sinusoidal wave) and one output (the IF signal). A local oscillator is used to generate a sinusoidal wave with a constant frequency (RF - IF). You may use a voltage controlled oscillator (VCO) for this purpose. At the output of the mixer, we get two chunks of spectrum centered at IF and 2 * RF - IF respectively. Therefore, the mixer should be followed by an LPF to remove the 2 * RF - IF part and leave the IF part. Finally, the IF signal could get into the ADC while obeying the Nyquist criteria. If you think these Minicircuits parts will make your experiment table messy, you could also purchase some fancy integrated modules such as the mc4020 cable modem tuner module. You are a USRP user, as we recommended, some new receiving daughter boards will bring us great convenience.

3 The Software

The digital signals finally get into the computer. What waits for them is our code - the so called software. GNU Radio provides a library of signal processing blocks and the glue to tie it all together. We will talk about the GNU Radio programming in detail in later chapters. It's worth to give it a shot now.

In GNU Radio, the programmer builds a radio by creating a graph (as in graph theory) where the vertices are signal processing blocks and the edges represent the data flow between them. The signal processing blocks are implemented in C++. Conceptually, blocks process infinite streams of data flowing from their input ports to their output ports. Blocks' attributes include the number of input and output ports they have as well as the type of data that flows through each. The most frequently used types are short, float and complex.

Some blocks have only output ports or input ports. These serve as data sources and sinks in the graph. There are sources that read from a file or ADC, and sinks that write to a file, digital-to-analog converter (DAC) or graphical display. About 100 blocks come with GNU Radio. Writing new blocks is not difficult.

Graphs are constructed and run in Python. Here is the 'Hello World' of GNU Radio. It generates two sine waves and outputs them to the sound card, one on the left channel, one on the right.

```
Hello World Example: Dial Tone Output

#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 48000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect ((src0, 0), (dst, 0))
    fg.connect ((src1, 0), (dst, 1))

    return fg
```

```
if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

We start by creating a flow graph to hold the blocks and connections between them. The two sine waves are generated by the `gr.sig_source_f` calls. The `f` suffix indicates that the source produces floats. One sine wave is at 350 Hz, and the other is at 440 Hz. Together, they sound like the US dial tone.

`audio.sink` is a sink that writes its input to the sound card. It takes one or more streams of floats in the range -1 to +1 as its input. We connect the three blocks together using the `connect()` method of the flow graph.

`connect()` takes two parameters, the source endpoint and the destination endpoint, and creates a connection from the source to the destination. An endpoint has two components: a signal processing block and a port number. The port number specifies which input or output port of the specified block is to be connected. In the most general form, an endpoint is represented as a python tuple like this: `(block, port_number)`. When `port_number` is zero, the block may be used alone.

These two expressions are equivalent:

```
    fg.connect ((src1, 0), (dst, 1))
    fg.connect (src1, (dst, 1))
```

Once the graph is built, we start it. Calling `start()` forks one or more threads to run the computation described by the graph and returns control immediately to the caller. In this case, we simply wait for any keystroke.

Don't worry if you still feel confused about some parts of the code. We will talk about them in depth later. At this stage, it's enough to understand the framework of a typical GNU Radio program.

Note that graphical user interfaces (GUI) for GNU Radio applications are also available, such as the soft oscillograph and the soft spectrum analyzer. They are built using wxPython. These nice GUI tools add lustre to the GNU Radio system, which also bring us great convenience and flexibility.

4 The Hardware

GNU Radio is reasonably hardware-independent. Today's commodity multi-gigahertz, super-scalar CPUs with single-cycle floating-point units mean that serious digital signal processing is possible on the desktop. A 3 GHz Pentium or Athlon can evaluate 3 billion floating-point FIR taps/s. We now can build, virtually all in software, communication systems unthinkable only a few years ago.

Your computational requirements depend on what you're trying to do, but generally speaking, a 1 or 2 GHz machine with at least 256 MB of RAM should suffice. You also need some way to connect the analog world to your computer. Low-cost options include built-in sound cards and audiophile quality 96 kHz, 24-bit, add-in cards.

Regarding the RF front end and AD/DA converters, there are certainly many choices. However, the Universal Software Radio Peripheral (USRP) board, which is designed wholly for GNU Radio, is strongly recommended. In fact, most GNU Radio players are using USRP boards. Not only because they are nice and convenient to use, but also because you can get the most technical supports from the GNU Radio community. So, to make your life simpler, just choose the USRP. In next tutorial, we will investigate what happens on the USRP board.

5 Conclusion

Software radio is an exciting field, and GNU Radio provides the tools to start exploring. GNU Radio is definitely a nice system. Welcome to the GNU Radio world!

References

[1] Eric Blossom, **Exploring GNU Radio**,
http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

Tutorial 3: Entering the World of GNU Software Radio

Dawei Shen[1]

August 3, 2005

Abstract

This article provides an overview of the GNU Radio toolkit for building software radios. This tutorial is a modified version of Eric's article `*Exploring GNU Radio*'. The concepts of software defined radio and the basics of signal processing will be introduced. We are going to enter the exciting world of GNU Software Radio!

Contents

1 Introduction to GNU Software Radio

Software radio is the technique of getting the code as close to the antenna as possible. It turns radio hardware problems into software problems. The fundamental characteristic of software radio is that software defines the transmitted waveforms, and software demodulates the received waveforms. This is in contrast to most radios in which the processing is done with either analog circuitry or analog circuitry combined with digital chips.

Software radio is a revolution in radio design due to its ability to create radios that change on the fly, creating new choices for users. At the baseline, software radios can do pretty much anything a traditional radio can do. The exciting part is the flexibility that software provides us. Controlling a computer, with necessary hardware supports, to play with radios should be easier, interesting and attractive.

GNU Radio is a free software toolkit for learning about, building and deploying software radios. GNU Radio provides a library of signal processing blocks and the glue to tie it all together. It is free and open source, it comes with complete source code so anyone can look and see how the system is built.

2 The structure of a software radio system

2.1 Block diagram

This is a typical RX path for a software radio:

```
Antenna -> Receive RF Front End -> ADC -> Software Code
```

This is a typical TX path for a software radio:

```
Software Code -> DAC -> Transmit RF Front End -> Antenna
```

To understand the software part of the radio, we first need to understand a bit about the associated hardware. Examining the receive path in the figure, we see an antenna, a mysterious RF front end, an analog-to-digital converter (ADC) and a bunch of code.

2.2 Analog to Digital Converter (ADC)

The analog-to-digital converter is the bridge between the physical world of continuous analog signals and the world of discrete digital samples manipulated by software.

ADCs have two primary characteristics, sampling rate and dynamic range. Sampling rate is the number of times per second that the ADC measures the analog signal. Dynamic range refers to the difference between the smallest and largest signal that can be distinguished; it's a function of the number of bits in the ADC's digital output and the design of the converter. For example, an 8-bit converter at most can represent 256 (28) signal levels, while a 16-bit converter represents up to 65,536 levels. Generally speaking, device physics and cost impose trade-offs between the sample rate and dynamic range.

After getting through the ADC, the continuous signal becomes a sequence of numbers entering the computer, which can be processed digitally as an array in the software. So what kind of ADC could be a good choice if we want to play with GNU Radio? The best answer should be the USRP board, which is developed by Matt and Eric. We will talk about the USRP board later.

2.3 The RF Front End

To understand the role of the RF front end, we need to talk about a bit of theory. Nyquist theorem tells us that, to avoid aliasing when converting from analog to digital, the ADC sampling frequency must be at least twice the maximum frequency of the signal of interest, in order to sustain all the spectrum information accurately. Aliasing is what makes the wagon wheels look like they're going backward in the old westerns: the sampling rate of the movie camera is not fast enough to represent the position of the spokes unambiguously.

Assuming we're dealing with low pass signals - signals where the bandwidth of interest goes from 0 to f_{MAX}, the Nyquist criterion states that our sampling frequency needs to be at least 2 * f_{MAX}. But if our ADC runs at 20 MHz, how can we listen to broadcast FM radio at 92.1 MHz? The answer is the RF front end. The receive RF front end translates a range of frequencies appearing at its input (RF band) to a lower range at its output (IF band). For example, we could imagine an RF front end that translated the signals occurring in the 90 - 100 MHz range (RF) down to the 0 - 10 MHz range (IF).

Mostly, we can treat the RF front end as a black box with a single control, the center of the input range that's to be translated. As a concrete example, a cable modem tuner module translates a 6 MHz chunk of the spectrum centered between about 50 MHz and 800 MHz down to an output range centered at 5.75 MHz. The center frequency of the output range is called the intermediate frequency, or IF.

What can be used as the RF front end? If you like to build everything from scratch, you could purchase the Minicircuits parts for prototype. A typical structure is shown below:

```
Antenna -> Low Noise Amplifer (LNA) -> Low Pass Filter (LPF) -
> Mixer -> LPF ->ADC
                                            Local Oscillator -
> -^
```

The low noise amplifier (LNA) and the low pass filter (LPF) are used to select the bandwidth of interest and amplify the signal. For example, in order to receive the FM stations, you may like to use an LNA and an LPF with a cutoff frequency of 120MHz. A mixer can be treated as a multiplier, with two inputs (the RF signal and the local sinusoidal wave) and one output (the IF signal). A local oscillator is used to generate a sinusoidal wave with a constant frequency (RF - IF). You may use a voltage controlled oscillator (VCO) for this purpose. At the output of the mixer, we get two chunks of spectrum centered at IF and 2 * RF - IF respectively. Therefore, the mixer should be followed by an LPF to remove the 2 * RF - IF part and leave the IF part. Finally, the IF signal could get into the ADC while obeying the Nyquist criteria. If you think these Minicircuits parts will make your experiment table messy, you could also purchase some fancy integrated modules such as the mc4020 cable modem tuner module. You are a USRP user, as we recommended, some new receiving daughter boards will bring us great convenience.

3 The Software

The digital signals finally get into the computer. What waits for them is our code - the so called software. GNU Radio provides a library of signal processing blocks and the glue to tie it all together. We will talk about the GNU Radio programming in detail in later chapters. It's worth to give it a shot now.

In GNU Radio, the programmer builds a radio by creating a graph (as in graph theory) where the vertices are signal processing blocks and the edges represent the data flow between them. The signal processing blocks are implemented in C++. Conceptually, blocks process infinite streams of data flowing from their input ports to their output ports. Blocks' attributes include the number of input and output ports they have as well as the type of data that flows through each. The most frequently used types are short, float and complex.

Some blocks have only output ports or input ports. These serve as data sources and sinks in the graph. There are sources that read from a file or ADC, and sinks that write to a file, digital-to-analog converter (DAC) or graphical display. About 100 blocks come with GNU Radio. Writing new blocks is not difficult.

Graphs are constructed and run in Python. Here is the `Hello World' of GNU Radio. It generates two sine waves and outputs them to the sound card, one on the left channel, one on the right.

```
Hello World Example: Dial Tone Output

#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 48000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.
GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.
GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect ((src0, 0), (dst, 0))
    fg.connect ((src1, 0), (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

We start by creating a flow graph to hold the blocks and connections between them. The two sine waves are generated by the `gr.sig_source_f` calls. The `f` suffix indicates that the source produces floats. One sine wave is at 350 Hz, and the other is at 440 Hz. Together, they sound like the US dial tone.

`audio.sink` is a sink that writes its input to the sound card. It takes one or more streams of floats in the range -1 to +1 as its input. We connect the three blocks together using the `connect()` method of the flow graph.

`connect()` takes two parameters, the source endpoint and the destination endpoint, and creates a connection from the source to the destination. An endpoint has two components: a signal processing block and a port number. The port number specifies which input or output port of the specified block is to be connected. In the most general form, an endpoint is represented as a python tuple like this: `(block, port_number)`. When `port_number` is zero, the block may be used alone.

These two expressions are equivalent:

```
    fg.connect ((src1, 0), (dst, 1))
    fg.connect (src1, (dst, 1))
```

Once the graph is built, we start it. Calling `start()` forks one or more threads to run the computation described by the graph and returns control immediately to the caller. In this case, we simply wait for any keystroke.

Don't worry if you still feel confused about some parts of the code. We will talk about them in depth later. At this stage, it's enough to understand the framework of a typical GNU Radio program.

Note that graphical user interfaces (GUI) for GNU Radio applications are also available, such as the soft oscillograph and the soft spectrum analyzer. They are built using wxPython. These nice GUI tools add lustre to the GNU Radio system, which also bring us great convenience and flexibility.

4 The Hardware

GNU Radio is reasonably hardware-independent. Today's commodity multi-gigahertz, super-scalar CPUs with single-cycle floating-point units mean that serious digital signal processing is possible on the desktop. A 3 GHz Pentium or Athlon can evaluate 3 billion floating-point FIR taps/s. We now can build, virtually all in software, communication systems unthinkable only a few years ago.

Your computational requirements depend on what you're trying to do, but generally speaking, a 1 or 2 GHz machine with at least 256 MB of RAM should suffice. You also need some way to connect the analog world to your computer. Low-cost options include built-in sound cards and audiophile quality 96 kHz, 24-bit, add-in cards.

Regarding the RF front end and AD/DA converters, there are certainly many choices. However, the Universal Software Radio Peripheral (USRP) board, which is designed wholly for GNU Radio, is strongly recommended. In fact, most GNU Radio players are using USRP boards. Not only because they are nice and convenient to use, but also because you can get the most technical supports from the GNU Radio community. So, to make your life simpler, just choose the USRP. In next tutorial, we will investigate what happens on the USRP board.

5 Conclusion

Software radio is an exciting field, and GNU Radio provides the tools to start exploring. GNU Radio is definitely a nice system. Welcome to the GNU Radio world!

References

[1]
Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

---

Footnotes:

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

---

On 18 Aug 2005, 15:12.

Tutorial 4: The USRP Board

Dawei Shen[*]

August 8, 2005

Abstract

This article introduces the Universal Software Radio Peripheral (USRP) board, which is GNU Radio's hardware counterpart.

1 Introduction to the USRP board

These days, when we talk about the GNU Radio, the Universal Software Radio Peripheral (USRP) board has become an indispensable hardware component. It is developed by Matt wholly for the GNU Radio users. Basically, the USRP is an integrated board which incorporates AD/DA converters, some forms of RF front end, and an FPGA which does some important but computationally expensive pre-processing of the input signal. The USRP is low-cost and high speed, which is the best choice for a GNU Radio user to implement some real time applications. We could purchase the USRP boards from Ettus. A USRP board consists of one mother board and up to four daughter boards. The price for the mother board is $450 and basic daughterboards cost $50 each. Figure 1 shows the picture of a USRP board equipped with four daughter boards (2 for RX and 2 for TX).

2 A 'data sheet' of the USRP board

This section introduces the hardware parts on the USRP board. It should be emphasized that the characteristics of those hardware parts are very important. They will influence your radio design and software programming extensively. You have to follow the constraints imposed by the hardware and memorizing some of them would be useful. So please read this section carefully.

A typical setup of the USRP board consists of one mother board and up to four daughter boards, as shown in Figure 2. On the mother board, we can see the DC power input and the USB 2.0 interface. At this stage, USB 1.x is not supported at all. So make sure your PC is equipped with USB 2.0.

2.1 AD / DA Converters

There are 4 high-speed 12-bit AD converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz. The AD converters can bandpass-sample signals

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

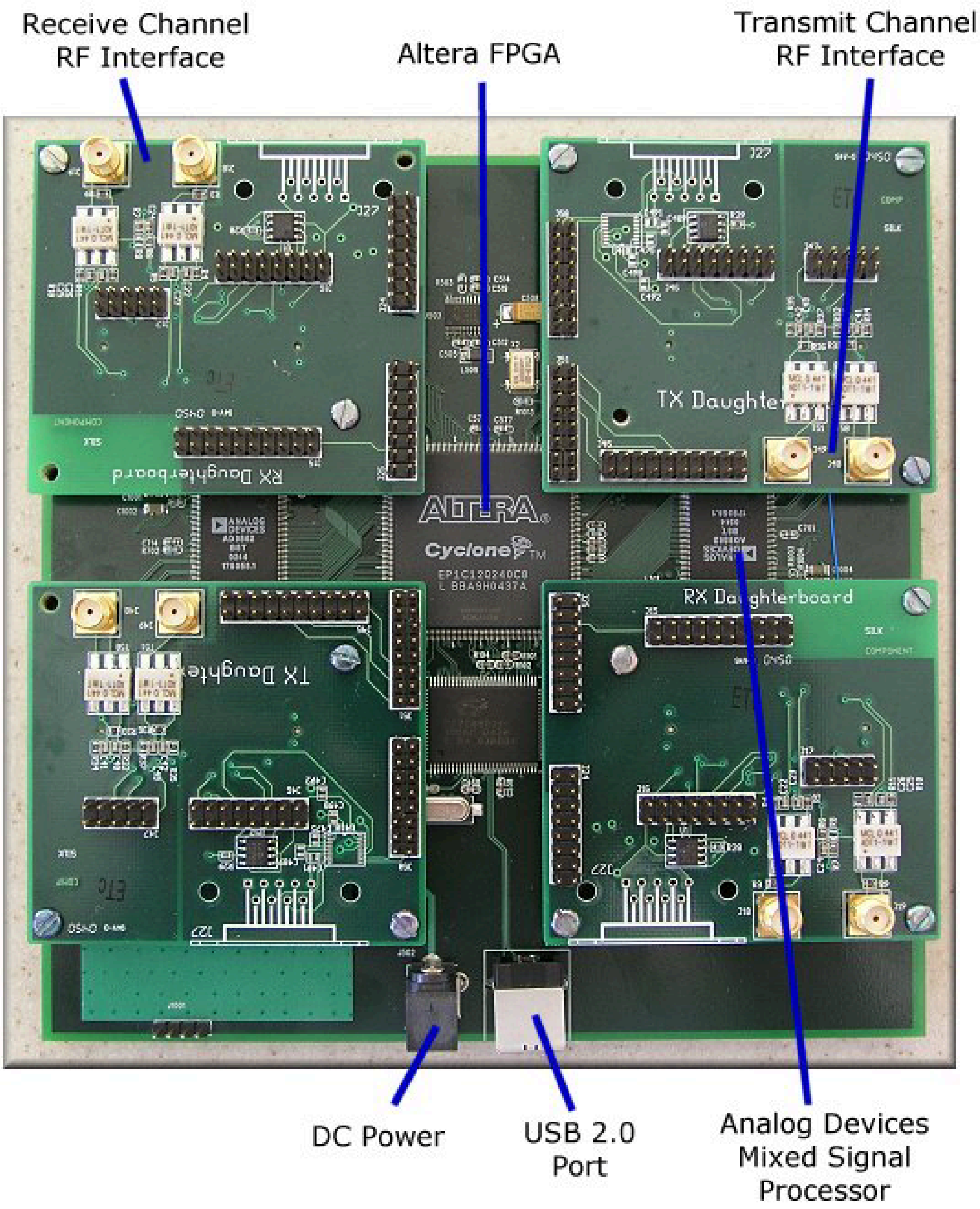
Receive Channel RF Interface
Altera FPGA
Transmit Channel RF Interface
DC Power
USB 2.0 Port
Analog Devices Mixed Signal Processor

Figure 1: The USRP board: One Mother board, two RX / two TX daughter boards

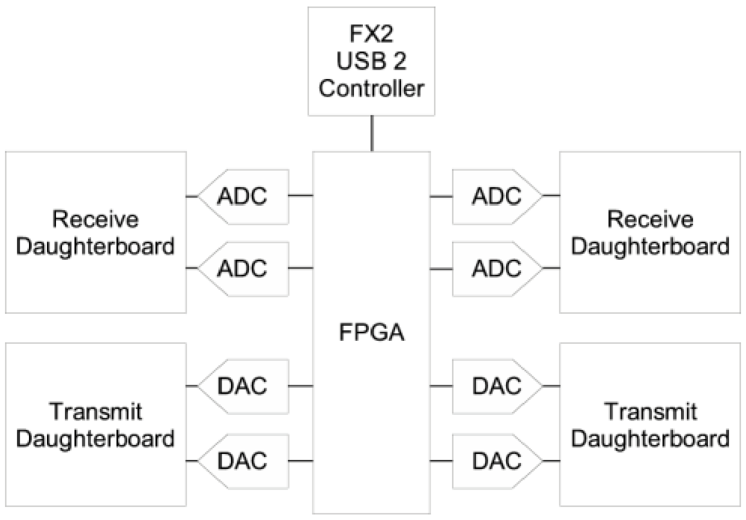

Figure 2: The USRP board

of up to about 150MHz, though. If we sample a signal with the IF larger than 32MHz, we introduce aliasing and actually the band of the signal of interest is mapped to some places between -32MHz and 32MHz. Sometimes this can be useful, for example, we could listen to the FM stations without any RF front end. The higher the frequency of the sampled signal, the more the SNR will be degraded by jitter. 100MHz is the recommended upper limit.

The full range on the ADCs is 2V peak to peak, and the input is 50 ohms differential. This is 40mW, or 16dBm. There is a programmable gain amplifier (PGA) before the ADCs to amplify the input signal to utilize the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20dB. Note that we can use other sampling rates if desired. The available rates are all submultiples of 128MHz, such as 64 MS/s, 42.66 MS/s, 32 MS/s, 25.6 MS/s and 21.33 MS/s.

At the transmitting path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, we will probably want to stay below about 50MHz or so to make filtering easier. So a useful output frequency range is DC to about 50MHz. The DACs can supply 1V peak to a 50 ohm differential load, or 10mW (10dBm). There is also PGA used after the DAC, providing up to 20dB gain. Note that the PGAs on both RX and TX paths are programmable.

So in principle, we have 4 input and 4 output channels if we use real samplings. However, we have more flexibility (and bandwidth) if we use complex (IQ) sampling. Then we have to pair them up, so we get 2 complex inputs and 2 complex outputs.

2.2 The daughter boards

On the mother board there are four slots, where you can plug in up to 2 RX daughter boards and 2 TX daughter boards. The daughter boards are used to hold the the RF receiver interface or tuner and the RF transmitter.

There are slots for 2 TX daughter boards, labeled TXA and TXB, and 2 corresponding RX daughter boards, RXA and RXB. Each daughter board slot has access to 2 of the 4 high-speed AD / DA converters (DAC outputs for TX, ADC inputs for RX). This allows each daughter board which uses real (not IQ) sampling to have 2 independent RF sections, and 2 antennas (4 total for the system). If complex IQ sampling is used, each board can support a single RF section, for a total of 2 for the whole system.

We can see there are two SMA connectors on each daughter board. We normally use them to connect the input or output signals.

There several kinds of daughter boards available now:

Basic daughter boards. Nothing fancy on it. Two SMA connectors are used to connect external tuners or signal generators. We can treat it as an entrance or an exit for the signal without affecting it. Some form of external RF front end is required.

TVRX daughter boards. With Microtune 4937 Cable Modem tuner equipped. This is a receive-only daughter board. The RF frequency ranges from 50MHz to 800MHz, with an IF bandwidth of 6MHz. What you need is just an antenna if your radio application is within this range, such as FM or TV detection.

DBSRX daughter boards. Similar with the TVRX boards. This is also receive-only. The RF frequency ranges from 800MHz to 2.4GHz.

Some new daughter boards are being developed, especially including the tranceiver daughter boards. They will be on sale soon and will bring us great convenience.

2.3 The FPGA

Probably understanding what goes on the FPGA is the most important part for GNU Radio users. As shown in Figure 2, all the ADCs and DACs are connected to the FPGA. This piece of FPGA plays a key role in the GNU Radio system. Basically what it does is to perform high bandwidth math, and to reduce the data rates to something you can squirt over USB2.0. The FPGA connects to a USB2 interface chip, the Cypress FX2. Everything (FPGA circuitry and USB Microcontroller) is programmable over the USB2 bus.

Our standard FPGA configuration includes digital down converters (DDC) implemented with cascaded integrator-comb (CIC) filters. CIC filters are very high-performance filters using only adds and delays. The FPGA implements 4 digital down converters (DDC). This allows 1, 2 or 4 separate RX channels. At the RX path, we have 4 ADCs, and 4 DDCs. Each DDC has two inputs I and Q. Each of the 4 ADCs can be routed to either of I or the Q input of any of the 4 DDCs. This allows for having multiple channels selected out of the same ADC sample stream.

The digital up converters (DUCs) on the transmit side are actually contained in the AD9862 CODEC chips, not in the FPGA. The only transmit signal processing blocks in the FPGA are the interpolators. The interpolator outputs can be routed to any of the 4 CODEC inputs.

The multiple RX channels (1,2, or 4) must all be the same data rate (i.e. same decimation ratio). The same applies to the 1,2, or TX channels, which each must be at the same data rate (which may be different from the RX rate).

Figure 3 shows the block diagram of the USRP's receive path and the diagram of the digital down converter. The MUX is like a router or a circuit switcher. It determines which ADC (or constant zero) is connected to each DDC input. There are 4 DDCs. Each has two inputs. We can control the MUX using `usrp.set_mux()` method in Python. We will talk about the GNU Radio programming in later tutorials, but it's worth to take a look at it now.

```
Mux value:

   3                   2                   1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+
|   Q3  |   I3  |   Q2  |   I2  |   Q1  |   I1  |   Q0  |   I0  |
+-------+-------+-------+-------+-------+-------+-------+-------+
```

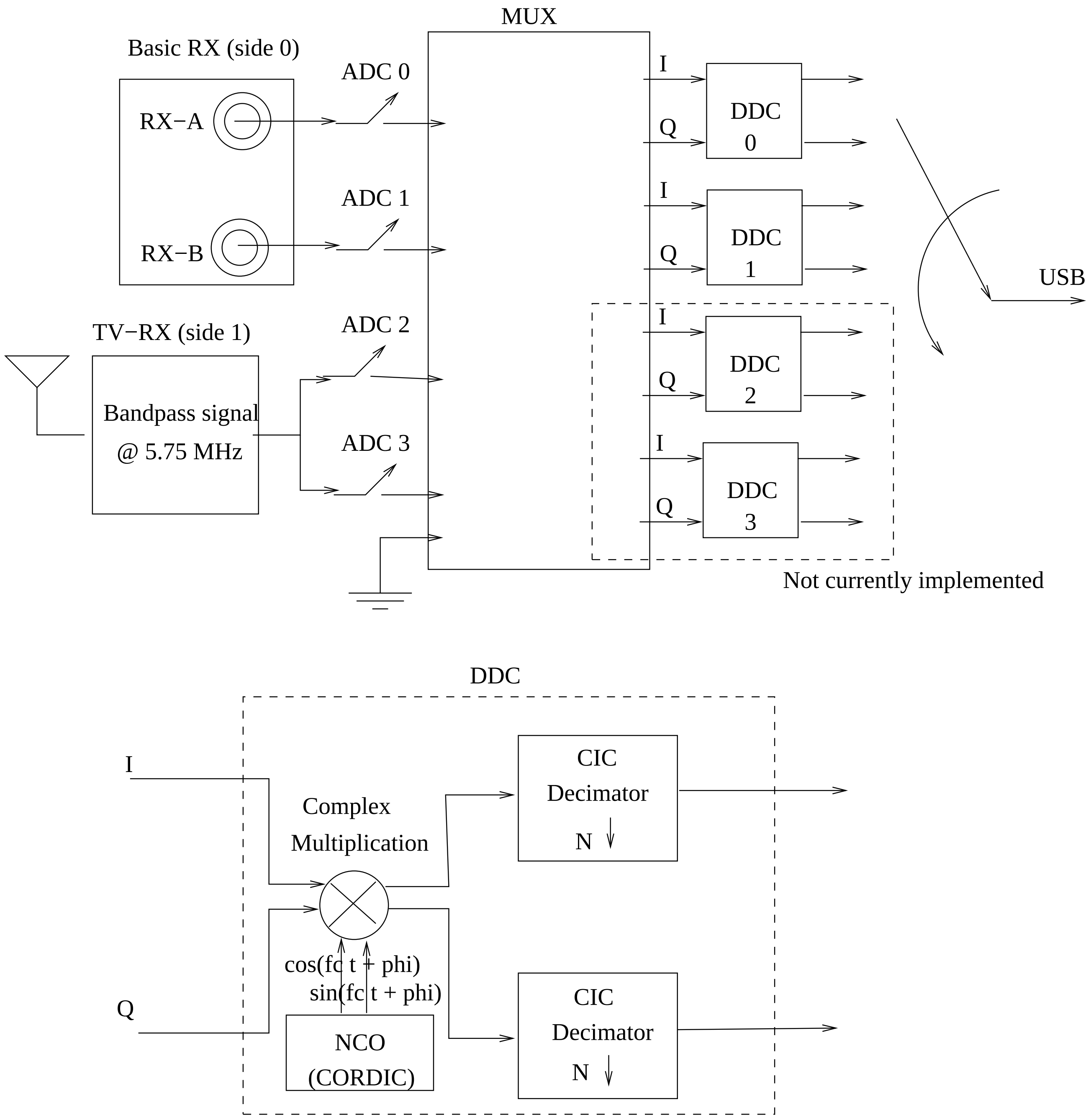

Figure 3: The block diagram of the USRP receive path

```
Each 4-bit I field is either 0,1,2,3
Each 4-bit Q field is either 0,1,2,3 or 0xf (input is const zero)
All Q's must be 0xf or none of them may be 0xf
```

We tell each input (I0, Q0, I1 ... I3, Q3) which ADC is connected to it by using 4 bits (0, 1, 2, 3 or 0xf). So a 32-bit integer would be enough for all 8 inputs to know which ADC is connected. Of course an integer in hexadecimal system will be more convenient if we want to use the `set_mux()` method. For most real sampling applications, the Q input of each DDC is constant zero. So quite often we don't need to modify the standard configuration of the FPGA. Actually it is anticipated that the majority of USRP users will never need to use anything other than the standard FPGA configuration.

Now let's see the digital down converter (DDC). What does it do? First, it down converts the signal from the IF band to the base band. Second, it decimates the signal so that the data rate can be adapted by the USB 2.0 and is reasonable for the computers' computing capability. The second part of Figure 3 shows the block diagram of the DDC. The complex input signal (IF) is multiplied by the constant frequency (usually also IF) exponential signal. The resulting signal is also complex and centered at 0. Then we decimate the signal with a factor N.

The decimator can be treated as a low pass filter followed by a downsampler. Suppose the decimation factor is D. If we look at the digital spectrum, the low pass filter selects out the band [$-\pi/D$, π/D], and then the downsampler spread the spectrum in [$-\pi/D$, π/D] to [$-\pi$, π]. So in fact, we have narrowed the bandwidth of the digital signal of interest by a factor of D. Regarding the bandwidth, we can sustain 32MB/sec across the USB. All samples sent over the USB interface are in 16-bit **signed integers** in IQ format, i.e. 16-bit I and 16-bit Q data (complex), resulting in 8M complex samples/sec across the USB. This provides a maximum effective total spectral bandwidth of about 8MHz by Nyquist criteria. Of course we can select much narrower ranges by changing the decimation rate. For example, suppose we want to design an FM receiver. The bandwidth of a FM station is generally 200kHz. So we can select the decimation factor to be 250. Then the data rate across the USB is 64MHz / 250 = 256kHz, which is well suited for the 200kHz bandwidth without losing any spectral information. We can set the IF frequency of the DDC using `usrp.set_rx_freq()` method and set the decimation factor using `usrp.set_decim_rate()` method in Python. The decimation rate must be in `[1, 256]`.

Note that when there are multiple channels (up to 4), the channels are interleaved. For example, with 4 channels, the sequence sent over the USB would be I0 Q0 I1 Q1 I2 Q2 I3 Q3 I0 Q0 I1 Q1, etc. The USRP can operate in full duplex mode. When in this mode, the transmit and receive sides are completely independent of one another. The only consideration is that the combined data rate over the bus must be 32 Megabytes per second or less.

OK! Finally the I/Q complex signal enters the computer via the USB. That's the software world!

At the TX path, the story is pretty much the same, except that it happens reversely. We need to send a baseband I/Q complex signal to the USRP board. The digital up converter (DUC) will interpolate the signal, up convert it to the IF band and finally send it through the DAC.

2.4 USRP Setup

There is almost nothing you need to do when you have received your new USRP boards. Probably the only thing you should do is to install all GNU Radio software packages correctly first, especially the `gnuradio-core`, `usrp` and `gr.usrp` packages. Please refer to tutorial 1 for the installation guide. Then you should be able to use the USRP board without any problem. Of course, you should make sure the DC supply, USB2.0 cable and daughter boards are properly connected.

When you are designing USRP related applications, you need USB device permissions to run your Python program. If you are the owner of your computer, maybe the simplest way is to put yourself in the `sudoers`.

Plug the Power cord into the usrp. You should see a green LED (hidden under the "RXA" daughter board, which is nearest to the USB connector) blinking at about 2Hz (twice a second). When a USRP application starts, the USRP will load two files from the directory `/usr/local/share/usrp/rev2`: `usrp_firmware.ihx` and `usrp_fpga.rbf`. The blinking LED should slow down to 1Hz (once a second) after the `usrp_firmware.ihx` file was loaded successfully. Ensure the file `usrp_fpga.rbf` is in that `/usr/local/share/usrp/rev2` directory. Note that if you install the `usrp` module from CVS, it won't be there. You'll have to get the tarball of usrp-XXX.tar.gz from the GNU Radio download page, extract this one file, copy it to the right place.

```
cp usrp-*.*/fpga/rbf/usrp_fpga_rev2.rbf  /usr/local/share/usrp/rev2/usrp_fpga.rbf
```

Finally let's try an example to test everything, including the software installation and the USRP. Find the `gnuradio-examples` directory, and go down into the `/python/usrp` subdirectory. Run

```
sudo ./usrp_oscope.py.
```

This should bring up something that looks like an oscilloscope, assuming you have installed `wxPython` and `gr-wxgui` correctly.

Then connect your speaker to the sound card and insert a piece of wire to the RX-A connector. Run the example:

```
sudo ./wfm_rcv_gui.py 101.5.
```

You should be able to hear strong FM stations.

If that works, then you have left the software installation and USRP setup stage behind you. Now you can use this exciting GNU Radio system to design your radio applications!

3 Conclusion

This article is actually a 'squeezed' version of the references [1][2][3], which covers almost all necessary information for most USRP users. Some details which may not be useful to most users are not introduced.

So far, we have completed the introduction to the software and hardware setup for GNU Radio. From next tutorial, we will focus on the GNU Radio programming.

References

[1] **Universal Software Radio Peripheral on Wiki**,
http://comsec.com/wiki?UniversalSoftwareRadioPeripheral

[2] **Rf Sections 4 USRP on Wiki**,
http://comsec.com/wiki?RfSections4USRP

[3] Matt Ettus **USRP User's and Developer's Guide**,
http://home.ettus.com/usrp/usrp_guide.html

[4] **USRP Install on Wiki**,
http://comsec.com/wiki?UsrpInstall

Tutorial 4: The USRP Board

Dawei Shen[1]

August 8, 2005

Abstract

This article introduces the Universal Software Radio Peripheral (USRP) board, which is GNU Radio's hardware counterpart.

Contents

1 Introduction to the USRP board

These days, when we talk about the GNU Radio, the Universal Software Radio Peripheral (USRP) board has become an indispensable hardware component. It is developed by Matt wholly for the GNU Radio users. Basically, the USRP is an integrated board which incorporates AD/DA converters, some forms of RF front end, and an FPGA which does some important but computationally expensive pre-processing of the input signal. The USRP is low-cost and high speed, which is the best choice for a GNU Radio user to implement some real time applications. We could purchase the USRP boards from Ettus. A USRP board consists of one mother board and up to four daughter boards. The price for the mother board is $450 and basic daughterboards cost $50 each. Figure 1 shows the picture of a USRP board equipped with four daughter boards (2 for RX and 2 for TX).

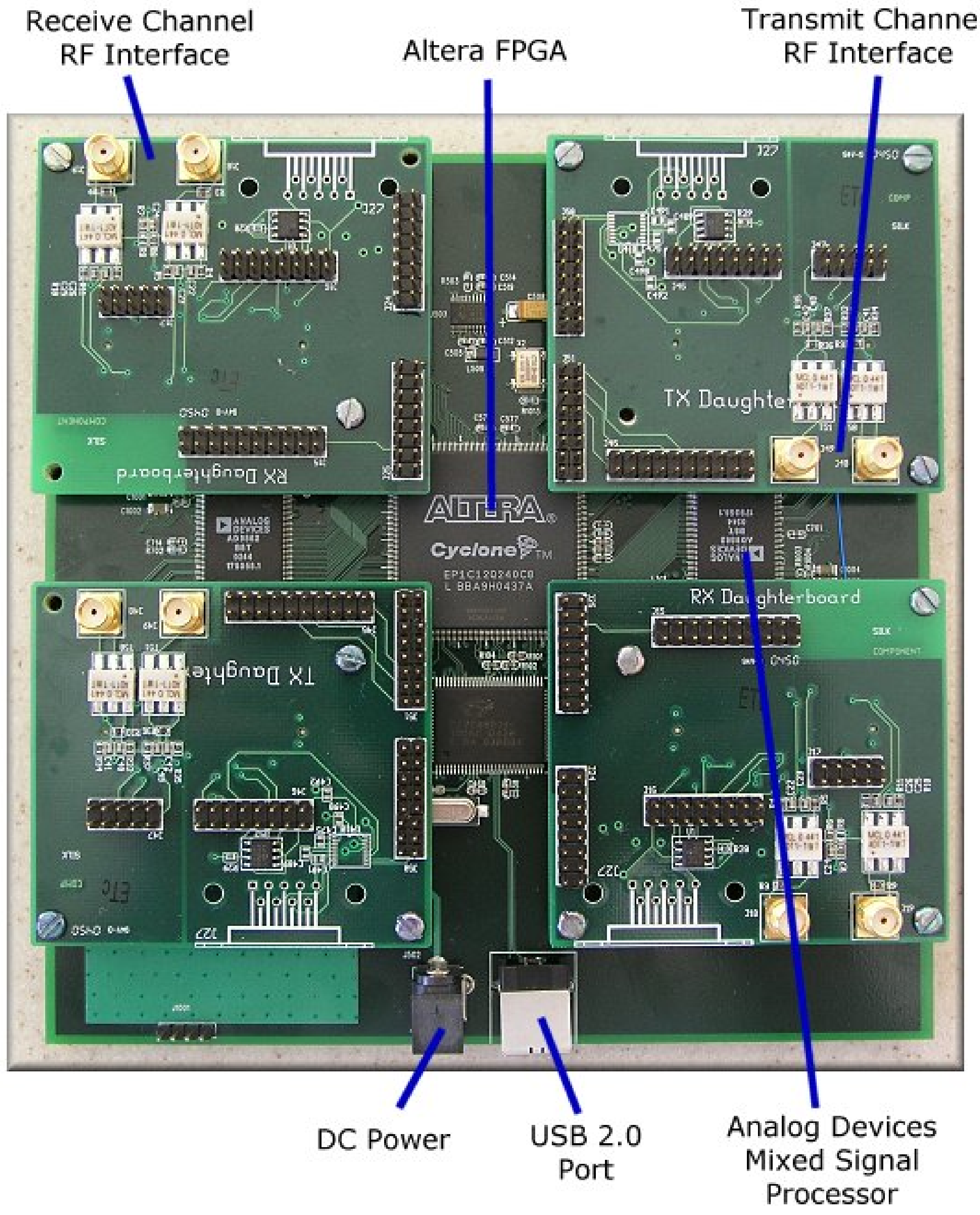
Receive Channel RF Interface
Altera FPGA
Transmit Channel RF Interface
RX Daughterboard
TX Daughter
TX Daughter
RX Daughterboard
DC Power
USB 2.0 Port
Analog Devices Mixed Signal Processor

Figure 1: The USRP board: One Mother board, two RX / two TX daughter boards

2 A `data sheet' of the USRP board

This section introduces the hardware parts on the USRP board. It should be emphasized that the characteristics of those hardware parts are very important. They will influence your radio design and software programming extensively. You have to follow the constraints imposed by the hardware and memorizing some of them would be useful. So please read this section carefully.

A typical setup of the USRP board consists of one mother board and up to four daughter boards, as shown in

Figure 2.

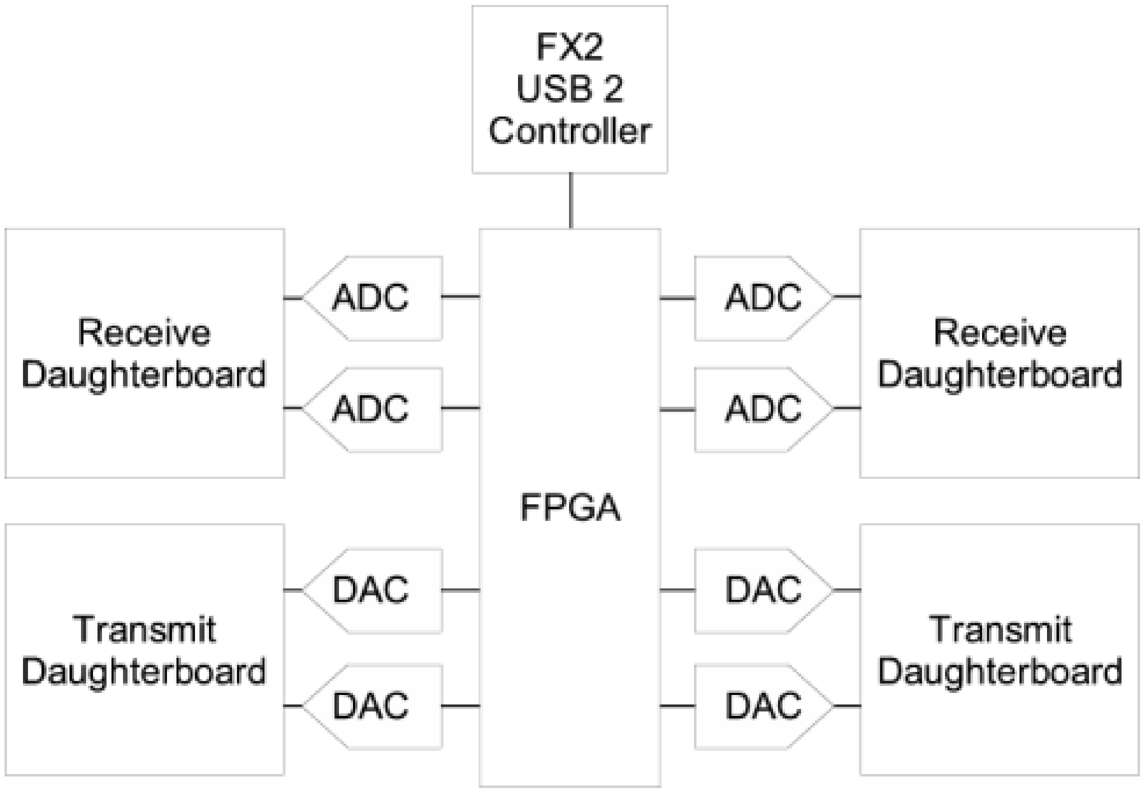

Figure 2: The USRP board

On the mother board, we can see the DC power input and the USB 2.0 interface. At this stage, USB 1.x is not supported at all. So make sure your PC is equipped with USB 2.0.

2.1 AD / DA Converters

There are 4 high-speed 12-bit AD converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz. The AD converters can bandpass-sample signals of up to about 150MHz, though. If we sample a signal with the IF larger than 32MHz, we introduce aliasing and actually the band of the signal of interest is mapped to some places between -32MHz and 32MHz. Sometimes this can be useful, for example, we could listen to the FM stations without any RF front end. The higher the frequency of the sampled signal, the more the SNR will be degraded by jitter. 100MHz is the recommended upper limit.

The full range on the ADCs is 2V peak to peak, and the input is 50 ohms differential. This is 40mW, or 16dBm. There is a programmable gain amplifier (PGA) before the ADCs to amplify the input signal to utilize the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20dB. Note that we can use other sampling rates if desired. The available rates are all submultiples of 128MHz, such as 64 MS/s, 42.66 MS/s, 32 MS/s, 25.6 MS/s and 21.33 MS/s.

At the transmitting path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, we will probably want to stay below about 50MHz or so to make filtering easier. So a useful output frequency range is DC to about 50MHz. The DACs can supply 1V peak to a 50 ohm differential load, or 10mW (10dBm). There is also PGA used after the DAC, providing up to 20dB gain. Note that the PGAs on both RX and TX paths are programmable.

So in principle, we have 4 input and 4 output channels if we use real samplings. However, we have more flexibility (and bandwidth) if we use complex (IQ) sampling. Then we have to pair them up, so we get 2 complex inputs and 2 complex outputs.

2.2 The daughter boards

On the mother board there are four slots, where you can plug in up to 2 RX daughter boards and 2 TX daughter boards. The daughter boards are used to hold the the RF receiver interface or tuner and the RF transmitter.

There are slots for 2 TX daughter boards, labeled TXA and TXB, and 2 corresponding RX daughter boards, RXA and RXB. Each daughter board slot has access to 2 of the 4 high-speed AD / DA converters (DAC outputs for TX, ADC inputs for RX). This allows each daughter board which uses real (not IQ) sampling to have 2 independent RF sections, and 2 antennas (4 total for the system). If complex IQ sampling is used, each board can support a single RF section, for a total of 2 for the whole system.

We can see there are two SMA connectors on each daughter board. We normally use them to connect the input or output signals.

There several kinds of daughter boards available now:

Basic daughter boards. Nothing fancy on it. Two SMA connectors are used to connect external tuners or signal generators. We can treat it as an entrance or an exit for the signal without affecting it. Some form of external RF front end is required.

TVRX daughter boards. With Microtune 4937 Cable Modem tuner equipped. This is a receive-only daughter board. The RF frequency ranges from 50MHz to 800MHz, with an IF bandwidth of 6MHz. What you need is just an antenna if your radio application is within this range, such as FM or TV detection.

DBSRX daughter boards. Similar with the TVRX boards. This is also receive-only. The RF frequency ranges from 800MHz to 2.4GHz.

Some new daughter boards are being developed, especially including the tranceiver daughter boards. They will be on sale soon and will bring us great convenience.

2.3 The FPGA

Probably understanding what goes on the FPGA is the most important part for GNU Radio users. As shown in Figure 2, all the ADCs and DACs are connected to the FPGA. This piece of FPGA plays a key role in the GNU Radio system. Basically what it does is to perform high bandwidth math, and to reduce the data rates to something you can squirt over USB2.0. The FPGA connects to a USB2 interface chip, the Cypress FX2. Everything (FPGA circuitry and USB Microcontroller) is programmable over the USB2 bus.

Our standard FPGA configuration includes digital down converters (DDC) implemented with cascaded integrator-comb (CIC) filters. CIC filters are very high-performance filters using only adds and delays. The FPGA implements 4 digital down converters (DDC). This allows 1, 2 or 4 separate RX channels. At the RX path, we have 4 ADCs, and 4 DDCs. Each DDC has two inputs I and Q. Each of the 4 ADCs can be routed to either of I or the Q input of any of the 4 DDCs. This allows for having multiple channels selected out of the same ADC sample stream.

The digital up converters (DUCs) on the transmit side are actually contained in the AD9862 CODEC chips, not in the FPGA. The only transmit signal processing blocks in the FPGA are the interpolators. The interpolator outputs can be routed to any of the 4 CODEC inputs.

The multiple RX channels (1,2, or 4) must all be the same data rate (i.e. same decimation ratio). The same applies to the 1,2, or TX channels, which each must be at the same data rate (which may be different from the RX rate).

Figure 3 shows the block diagram of the USRP's receive path and the diagram of the digital down converter.

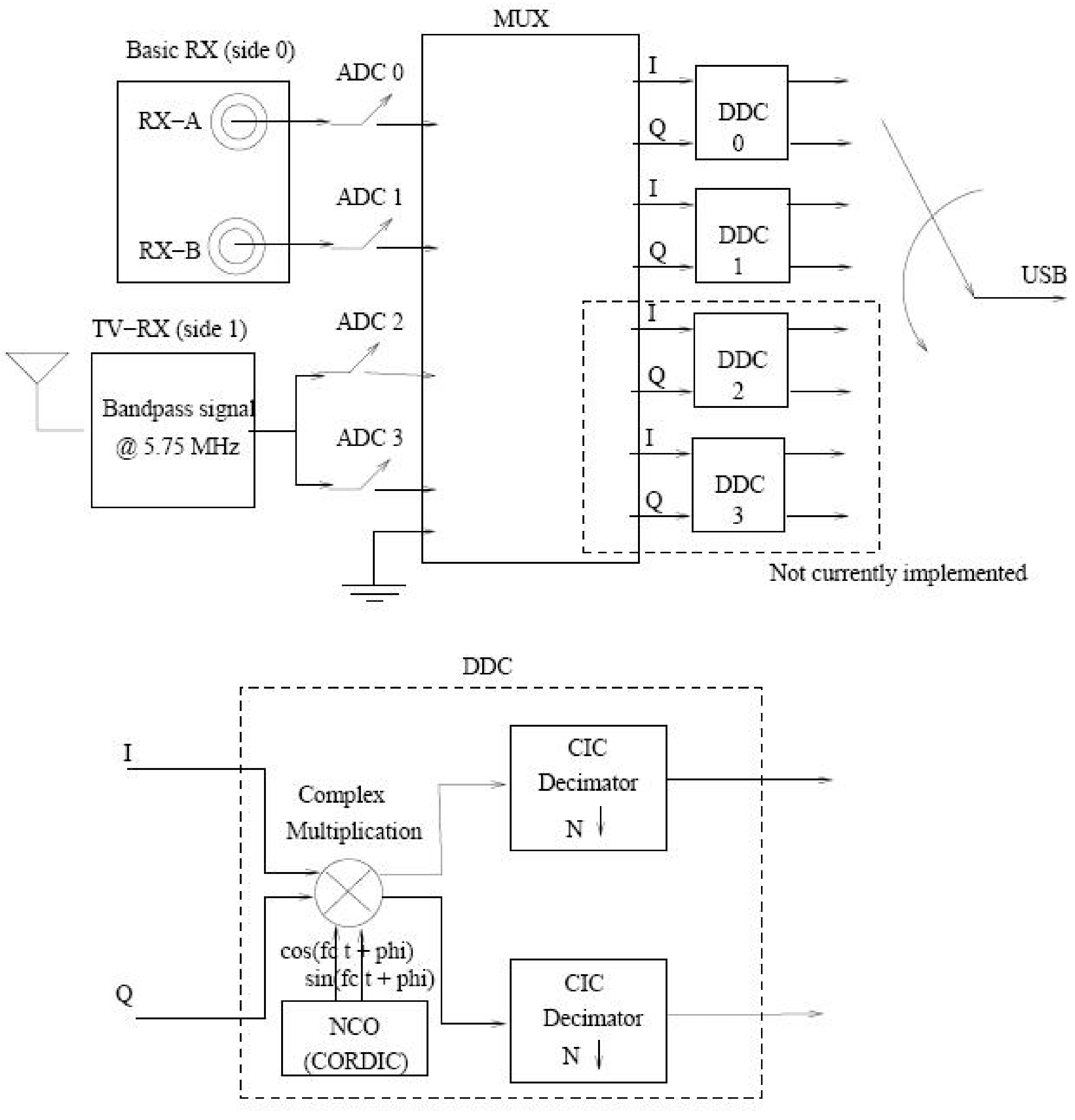

Figure 3: The block diagram of the USRP receive path

Figure 3: The block diagram of the USRP receive path

The MUX is like a router or a circuit switcher. It determines which ADC (or constant zero) is connected to each DDC input. There are 4 DDCs. Each has two inputs. We can control the MUX using `usrp.set_mux()` method in Python. We will talk about the GNU Radio programming in later tutorials, but it's worth to take a look at it now.

```
Mux value:

   3                   2                   1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+
|   Q3  |   I3  |   Q2  |   I2  |   Q1  |   I1  |   Q0  |   I0  |
+-------+-------+-------+-------+-------+-------+-------+-------+

Each 4-bit I field is either 0,1,2,3
Each 4-bit Q field is either 0,1,2,3 or 0xf (input is const zero)
All Q's must be 0xf or none of them may be 0xf
```

We tell each input (I0, Q0, I1 ... I3, Q3) which ADC is connected to it by using 4 bits (0, 1, 2, 3 or 0xf). So a 32-bit integer would be enough for all 8 inputs to know which ADC is connected. Of course an integer in hexadecimal system will be more convenient if we want to use the `set_mux()` method. For most real sampling applications, the Q input of each DDC is constant zero. So quite often we don't need to modify the standard configuration of the FPGA. Actually it is anticipated that the majority of USRP users will never need to use anything other than the standard FPGA configuration.

Now let's see the digital down converter (DDC). What does it do? First, it down converts the signal from the IF band to the base band. Second, it decimates the signal so that the data rate can be adapted by the USB 2.0 and is reasonable for the computers' computing capability. The second part of Figure 3 shows the block diagram of the DDC. The complex input signal (IF) is multiplied by the constant frequency (usually also IF) exponential signal. The resulting signal is also complex and centered at 0. Then we decimate the signal with a factor N.

The decimator can be treated as a low pass filter followed by a downsampler. Suppose the decimation factor is D. If we look at the digital spectrum, the low pass filter selects out the band $[-\pi/D, \pi/D]$, and then the downsampler spread the spectrum in $[-\pi/D, \pi/D]$ to $[-\pi, \pi]$. So in fact, we have narrowed the bandwidth of the digital signal of interest by a factor of D. Regarding the bandwidth, we can sustain 32MB/sec across the USB. All samples sent over the USB interface are in 16-bit **signed integers** in IQ format, i.e. 16-bit I and 16-bit Q data (complex), resulting in 8M complex samples/sec across the USB. This provides a maximum effective total spectral bandwidth of about 8MHz by Nyquist criteria. Of course we can select much narrower ranges by changing the decimation rate. For example, suppose we want to design an FM receiver. The bandwidth of a FM station is generally 200kHz. So we can select the decimation factor to be 250. Then the data rate across the USB is 64MHz / 250 = 256kHz, which is well suited for the 200kHz bandwidth without losing any spectral information. We can set the IF frequency of the DDC using `usrp.set_rx_freq()` method and set the decimation factor using `usrp.set_decim_rate` `() method in Python.  The decimation rate must be in` [1, 256].

Note that when there are multiple channels (up to 4), the channels are interleaved. For example, with 4 channels, the sequence sent over the USB would be I0 Q0 I1 Q1 I2 Q2 I3 Q3 I0 Q0 I1 Q1, etc. The USRP can operate in full duplex mode. When in this mode, the transmit and receive sides are completely independent of one another. The only consideration is that the combined data rate over the bus must be 32 Megabytes per second or less.

OK! Finally the I/Q complex signal enters the computer via the USB. That's the software world!

At the TX path, the story is pretty much the same, except that it happens reversely. We need to send a baseband I/Q complex signal to the USRP board. The digital up converter (DUC) will interpolate the signal, up convert it to the IF band and finally send it through the DAC.

2.4 USRP Setup

There is almost nothing you need to do when you have received your new USRP boards. Probably the only thing you should do is to install all GNU Radio software packages correctly first, especially the `gnuradio-core`, `usrp` and `gr.usrp` packages. Please refer to tutorial 1 for the installation guide. Then you should be able to use the USRP board without any problem. Of course, you should make sure the DC supply, USB2.0 cable and daughter boards are properly connected.

When you are designing USRP related applications, you need USB device permissions to run your Python program. If you are the owner of your computer, maybe the simplest way is to put yourself in the `sudoers`.

Plug the Power cord into the usrp. You should see a green LED (hidden under the "RXA" daughter board, which is nearest to the USB connector) blinking at about 2Hz (twice a second). When a USRP application starts, the USRP will load two files from the directory `/usr/local/share/usrp/rev2`: `usrp_firmware.ihx` and `usrp_fpga.rbf`. The blinking LED should slow down to 1Hz (once a second) after the `usrp_firmware.ihx` file was loaded successfully. Ensure the file `usrp_fpga.rbf` is in that `/usr/local/share/usrp/rev2` directory. Note that if you install the `usrp` module from CVS, it won't be there. You'll have to get the tarball of usrp-XXX.tar.gz from the GNU Radio download page, extract this one file, copy it to the right place.

```
cp usrp-*.*/fpga/rbf/usrp_fpga_rev2.rbf  /usr/local/share/usrp/
rev2/usrp_fpga.rbf
```

Finally let's try an example to test everything, including the software installation and the USRP. Find the `gnuradio-examples` directory, and go down into the `/python/usrp` subdirectory. Run

```
sudo ./usrp_oscope.py.
```

This should bring up something that looks like an oscilloscope, assuming you have installed `wxPython` and `gr-wxgui` correctly.

Then connect your speaker to the sound card and insert a piece of wire to the RX-A connector. Run the example:

```
sudo ./wfm_rcv_gui.py 101.5.
```

You should be able to hear strong FM stations.

If that works, then you have left the software installation and USRP setup stage behind you. Now you can use this exciting GNU Radio system to design your radio applications!

3 Conclusion

This article is actually a `squeezed' version of the references [1][2][3], which covers almost all necessary information for most USRP users. Some details which may not be useful to most users are not introduced.

So far, we have completed the introduction to the software and hardware setup for GNU Radio. From next tutorial, we will focus on the GNU Radio programming.

References

[1]

Universal Software Radio Peripheral on Wiki,
http://comsec.com/wiki?UniversalSoftwareRadioPeripheral

[2]

Rf Sections 4 USRP on Wiki,
http://comsec.com/wiki?RfSections4USRP

[3]

Matt Ettus **USRP User's and Developer's Guide**,
http://home.ettus.com/usrp/usrp_guide.html

[4]

USRP Install on Wiki,
http://comsec.com/wiki?UsrpInstall

Footnotes:

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

---

Tutorial 5: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line – Part I

Dawei Shen[*]

May 27, 2005

Abstract

Python plays a key role in GNU Radio programming. GNU Radio provides a framework for building software radios. The signal processing applications – are built using a combination of Python code for high level organization, policy, GUI and other non performance-critical functions, while performance critical signal processing blocks are written in C++. From the Python point of view, GNU Radio provides a data flow abstraction. This article is focused on the Python level, introducing the basic usage of Python and how Python is used in GNU Radio to glue all the signal processing blocks and control the flow of the digital data. A popular and classical example: the implementation of an FM receiver with GUI, is used here. The code is analyzed line by line. The basic grammar of Python and the concept of software radio are introduced in parallel. So this article can be used as both a short Python tutorial and a software radio guidance. How to write the blocks using C++ and other advanced topics will be covered in subsequent chapters.

---

1 Overview

GNU Radio's software is organized using a two-tier structure. All the performance-critical signal processing blocks are implemented in C++, while the higher-level organizing, connecting and gluing are done using Python. Many frequently used signal processing blocks have been implemented well and provided to us as parts of the GNU Radio software.

This structure has some similarity with the OSI 7-layer data network model. Lower layer provides service to the higher layer, while the higher layer doesn't care about the implementation details carried on in lower layers, but necessary interfaces and function calls. In GNU Radio, this layer transparency exists in a similar way. From the Python's point of view, what it does is just to select necessary signal sources, sinks and processing blocks, set correct parameters, then connect them together to form a complete application. In fact, all these sources, sinks and blocks are implemented as classes in C++. The parameter setting, connecting operations correspond to some sophisticated functions or class methods in C++. However, Python can't see how sedulously C++ has been working. A piece of lengthy, complicated and powerful C++ code is nothing but an interface to Python.

As a result, no matter how complicated the application is, the Python code is almost always short and neat. The real heavy load is thrown to C++. A thumb of rule should be kept in mind: for any application, what we need to do at the Python level, is always just to draw a diagram showing the signal flow from the source to the sink in your mind, then use the 'nice pen' −− Python, to find them and connect them all together, sometimes with the graphical user interfaces (GUI) support.

Obviously Python is crucial in learning GNU Radio. Python is a powerful and flexible programming language, which itself is a long story. But if you have enough C/C++ background, it's just a piece

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

of cake. Considering the fact that Python has some special characteristics when applied to GNU Radio and that some of its cool fancy features may not be necessary in GNU Radio, the article aims at combining the Python programming techniques and software radio concepts together. I am a believer that the best way to grasp the essence of some new knowledge is to go through an example rather than learning the syntax or semantics dryly. So our discussion will surround a popular and classical example: the implementation of an FM receiver with graphical user interfaces. The code will be analyzed line by line and the Python programming, signal processing techniques, software radio concepts and some hardware configuration will be talked about along the way.

This example implements an FM receiver with graphical user interface. The FM signal from the air is received by the USRP board then gets processed in the USRP board and in the computer. Finally the demodulated signal is played using the sound card. No fancy antenna is required. You can hear very high quality FM signal by just inserting a copper wire to the basic RX daughter board. The code can be found at '`gnuradio-examples/python/usrp/wfm_rcv_gui.py`'. Please refer to appendix A to make sure we have the same version of the code. We plan to use two articles to cover the whole materials. This article is the first half, focusing on the basics of Python and GNU Radio. Some advanced topics, such as the GUI tools (wxPython) and the usage of some Python built-in packages, are left to Tutorial 8.

2 The first line

If you have read the code of other examples, you can find the first line of these programs is almost always

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

The Python scripts can be made directly executable if we put this line at the beginning of the script and giving the file an executable mode. The '#!' must be the first two characters of the file. The script is given the executable mode by using the '**chmod**' command:

```
$ chmod +x wfm_rcv_gui.py
```

Now the script `wfm_rcv_gui.py` becomes executable. You can run this program in the shell using

```
$ ./wfm_rcv_gui.py arguments
```

the Python interpreter will be invoked and the code in this script will be executed line by line orderly. Python is an interpreted language, like Matlab script. No compilation and linking is necessary.

There are several ways to invoke the Python interpreter: you can use

```
$ python ./wfm_rcv_gui.py arguments
```

without the need to give the script the executable mode.

You can also use the interactive mode, by just typing the command in the shell:

```
$ python
```

then the Python interpreter environment will be invoked and you could input your code line by line. However, this is obviously inconvenient. We seldom use the interpreter interactively, unless we write some throw-away programs, test functions or use it as a desk calculator. Most of the time, packing codes in a **.py** file and make the script self-executable is more convenient to us.

3 Importing necessary modules

Next, we see a lot of importing stuffs:

```
from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
from gnuradio.wxgui import stdgui, fftsink
import wx
```

Understanding these statements requires the knowledge of '**module**' and '**package**' concepts in Python. The best way to learn them is to go through the Chapter 6 of the Python tutorials.

Here is a brief introduction. If we quit the Python interpreter and enter it again, all the functions and variables we have defined are lost. Therefore, we wish to write a somewhat longer program and save it as a **script**, containing function and variable definitions, and maybe some executive statements. This script can be used as the input of the Python interpreter. We may also want to use a fancy function we've written in several programs without copying its definitions to each program.

To support this, Python provides a **module/package** organization system. A **module** is a file containing Python definitions and statements, with the suffix '**.py**'. Within a module, the module's name (as a string) is available as the value of the global variable '`__name__`'. Definitions in a module can be imported into other modules or into the top-level module. A **package** is a collection of modules that have similar functions, which are often put in the same directory. The '`__init__.py`' files are required to make Python treat the directories as packages. A package could contain both modules and sub-packages (can contain sub-sub-packages). We use 'dotted module names' to structure Python's module namespace. For example, the module name A.B designates a submodule named 'B' in a package named 'A'.

A module can contain executable statements as well as function definitions. The statements are executed only the first time the module is imported somewhere and so that the module is initialized. Each module has its own private symbol table, which is used as the global symbol table by all functions defined within that module. The author of a module can use the global variables in the module without worrying about accidental clashes with the module user's global variables. As a module user, we can access a module's function and global variables using '**modname.itemname**'. Here the '**itemname**' can either be a function or a variable.

Modules can import other modules using '**import**' command. It is customary to place all '**import**' statements at the beginning of a module. Note that the **import** operation is quite flexible. We can import a package, a module, or just a definition within a module. When we try to import a module from a package, we can either use '**import packageA.moduleB**', or '**from package A import module B**'. When using '**from package import item**', the '**item**' can be either a module/sub-package of the package, or some other names defined in the package, like functions, classes or variables.

It's worth taking a while to introduce the modules used in this example amply, because these modules or packages will be frequently encountered in GNU Radio. The top-level package of GNU Radio is '*gnuradio*', which includes all GNU Radio related modules. It is located at

```
/usr/local/lib/python2.4/site-packages
```

By default, this directory is not included in the Python's search path, we need to export the path to the environment variable '**PATHONPATH**'. So we usually add the following line to the users' `.bash_profile` file:

```
export PATHONPATH=/usr/local/lib/python2.4/site-packages
```

to make sure the Python interpreter could find the *gnuradio* package.

gr is an important sub-package of *gnuradio*, which is the core of the GNU Radio software. The type of '**flow graph**' classes is defined in *gr* and it plays a key role in scheduling the signal flow.

eng_notation is a module designed for engineers' notation convenience, in which many words and characters are endowed with new constant values according to the engineering convention. The module *audio* provides the interfaces to access the sound card, while *usrp* provides the interfaces to control the USRP board. *audio* and *usrp* are often used as the signal source and sink. We will see the details about them later in this article. *blks* is a sub-package, which is almost an empty folder if you check its directory. It actually transfers all its tasks to another sub-package *blksimpl* in *gnuradio*, as described in the `__init__.py` file. *blksimpl* provides the implementation of several useful applications, such as FM receiver, GMSK, etc. For this example, the real signal processing part of the FM receiver is performed in this package.

Look at the next line, which is more interesting:

```
from gnuradio.eng_option import eng_option
```

This is exactly what we mentioned just now, we can either import a complete module/sub-package, or, just a function, class or variable definition from this module. In this case, *eng_option* is a class defined in the module *gnuradio.eng_option*. We don't need the whole module to be imported, but just a single class definition. *gnuradio.eng_option* module does nothing but adding support for engineering notation to *optparse.OptionParser*.

This line appears to have a similar format:

```
from gnuradio.wxgui import stdgui, fftsink
```

But the meaning is a little bit different, *gnuradio.wxgui* is a sub-package, not a module, while *stdgui* and *fftsink* are two modules in this sub-package. It's not necessary to import the whole sub-package, so we just import what we want explicitly. *gnuradio.wxgui* provides visualization tools for GNU Radio, which is constructed based on wxPython. The importing operations in Python provide us great flexibility and convenience.

Finally, *optparse*, *math*, *sys*, *wx* are all Python or wxPython's built-in modules or sub-packages, which are not part of the GNU Radio.

At this point, let me emphasize again, these modules imported above may contain executable statements as well as the function or class definitions. The statements will be executed immediately after the modules are imported. After importing the modules and packages, a lot of variables, classes and modules defined in them have been initialized. So don't think nothing has been done. Actually a lot of work behind the stage has been carried on. Many guys are waiting for your order in the workspace.

OK! So far we have taken a quick look at the most frequently used modules in GNU Radio and seen how they are organized together. Maybe it sounds too abstract and you are still confused about their usage. Never mind, we will see them soon.

4 The story in the class wfm_rx_graph

Familiarizing with object oriented programming (OOP) is important for understanding this section. OOP itself is a long story, which is obviously not what we are focusing on. But we will talk about some OOP concepts along the way. Some digital signal processing techniques will also be discussed as a reminder if we meet the corresponding codes.

4.1 Class definition

In this example, a large part of the code is the definition of a class '`wfm_rx_graph`'. The statement

```
class wfm_rx_graph (stdgui.gui_flow_graph):
```

defines a new class '`wfm_rx_graph`', which is inherited (derived) from the base class, or the so called 'father class' --'`gui_flow_graph`'. The father class `gui_flow_graph` is defined in the module *stdgui* we have just imported from *gnuradio*. By the rules of the namespace, we use `stdgui.gui_flow_graph` to refer to it.

4.2 The family of 'FLOW GRAPH' classes

Here an important category of classes, which play a key role in GNU Radio should receive particular attention: the 'flow graph' classes. There are a series of 'GRAPH' related classes defined in GNU Radio. We can find that `stdgui.gui_flow_graph` is derived from `gr.flow_graph`, which is defined in the sub-package *gr*. Further, `gr.flow_graph` is derived from the 'root' class `gr.basic_flow_class`. In GNU Radio, there are also many other classes derived from `gr.basic_flow_graph`. This big 'GRAPH family' makes GNU Radio programming neat and simple, also makes the scheduling of the signal processing clear and straightforward.

What do these 'graphs' do? Suppose you are trying to design a circuit using some commercial software such as Pspice. Probably you will first open a schematic, or a 'canvas', then you put all necessary circuit parts, such as a resistor, an amplifier or some DC powers on this canvas. Finally, you draw lines among these parts to connect them together and complete the circuit design. This scenario applies perfectly into GNU Radio. A **graph** is just like the schematic or the canvas. The circuit parts are replaced by signal sources, sinks and the signal processing blocks in GNU Radio. Finally, those 'wires' correspond to the '**connect**' method of the graph class, which is in charge of gluing these blocks together. Definitely, sometimes, an integrated circuit can serve as a part, the so called sub-circuit, in another schematic. That's also true in GNU Radio, a sub-graph can be used as a whole block in another graph.

In our example, `wfm_rx_graph` is such a graph class belonging to this family, with GUI support. Later we will see it glues the necessary blocks in FM receiver together using the method '**connect**'.

4.3 The initialization function: __init__

Then we implement the method (or function) '`__init__`' of the class `wfm_rx_graph`. The syntax for defining a new method is

```
def funcname(arg1 arg2 ...)
```

`__init__` is an important method for any class. After defining the class, we may use the class to instantiate an instance. This special method `__init__` is used to create an object in a known initial state. Class instantiation automatically invokes `__init__` for the newly created class instance. Actually in this example, `__init__` is the only method defined in the class `wfm_rx_graph`.

One important feature of Python is worth mentioning before we talk about the details in the function `__init__`. We notice that in this piece of code, there is no explicit signs for where a definition of a class or a function starts and ends. Usually in other programming languages such as C++ or Pascal, we use '**begin**' and '**end**' pair, or a pair of '{' and '}' explicitly to denote the two ends of a group of statements. However, in Python, this is no longer the case. There is **NO** such signs. In Python, statement grouping is done by **indentation** instead of beginning and ending brackets. So be careful about your editing and layout of the code when you write programs using Python.

Now let's see what's going on in the function `__init__`.

```
def __init__(self,frame,panel,vbox,argv):
```

declares the initialization method `__init__` with four arguments. Conventionally, the first argument of all methods are often called **self**. This is nothing more than a convention: the name **self** has absolutely no special meaning to Python. However, methods may call other methods by using method attributes of the **self** argument, such as '`self.connect()`' we will meet later.

The first thing `__init__` does, is to call the initialization method of `stdgui.gui_flow_graph`, its 'father class' , with exactly the same four arguments.

```
stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)
```

You may like to take a look at `stdgui.gui_flow_graph`'s initialization method. Since `wfm_rx_graph` is derived from it, we can safely think of `wfm_rx_graph` as a 'special' `gui_flow_graph`. So it's natural that this 'son class' should do something to make himself look like his father first, then do something fancy to make himself a different guy.

4.4 Constructing the graph with source, sink and signal processing blocks

4.4.1 Defining constants

From the next line, we kind of start to see the real signals coming in, which is less boring

```
IF_freq = parseargs(argv[1:])
```

sets the IF frequency from the return value of the function **parseargs**.

What does IF frequency stand for? IF is short for 'intermediate frequency'. Roughly speaking, it's the center frequency of the frequency band we are interested in. We move the 'real' frequency band, the so called RF band, which is usually very high, to some intermediate frequency band (IF) where the ADC can work obeying the Nyquist theorem. This isn't the key point in this article. I hope you have already got enough background in communications and DSP at this point.

The function **parseargs** is defined in this example later, right after the definition of the class `wfm_rx_graph`. It accepts the user's input arguments when the program is executed in shell. Let's talk about it later.

Besides, you may have noticed that Python doesn't require variable or argument declarations. This is totally different with the 'declare before use' concept in C.

```
adc_rate = 64e6
```

This line defines the sampling frequency of the AD converter, which should be set to 64MHz for the USRP users. According to Nyquist theorem, the maximum frequency component of the interested signal should be less than 32MHz in order not to loose spectrum information after sampling.

```
decim = 250
quad_rate = adc_rate / decim                # 256 kHz
```

Decimation is a concept in DSP world. After sampling the analog signal, we get a digital signal with very high data rate, which is a heavy burden for the CPU and storage. Usually, we can down-sample the digital sequence (decimation) without losing the spectrum information. In this example, the decimation rate is chosen to be 250 so that the resulting data rate is 256K samples per second, which is quite reasonable and acceptable for our CPU speed. `quad_rate` represents for **quadrature data rate**. The reason why it is called quadrature rate will be explained later.

'`# 256 kHz`' is nothing but a piece of comment. In Python, a comment starts after the symbol '#'. All statements after '#' in each line will be ignored by Python interpreter.

```
audio_decimation = 8
audio_rate = quad_rate / audio_decimation             # 32 kHz
```

After processing the digital FM signal, we wish to play the signal using the sound card of the computer. However, the data rate that a sound card can adopt is rather limited. 256kHz is usually too high and more than necessary. So we need to further decimate the data rate. 32kHz is a common choice for most sound cards.

4.4.2 The signal source

The following several lines provide a very high level abstraction for the signal processing procedure, which basically includes three components: the signal source, the signal sink, and a series of signal processing blocks. This example gives those signal processing blocks a very nice name: **guts**.

The signal source for the FM receiver is the USRP board in this example

```
# usrp is data source
src = usrp.source_c (0, decim)
src.set_rx_freq (0, IF_freq)
src.set_pga(0,20)
```

The USRP board receives the analog FM signal from the air via the RX daughter board and samples the signal using the AD converter with a sampling rate 64MHz. The resulting digital sequence then goes into the FPGA chip equipped on the USRP board. It is down-sampled there according to the decimation rate that the user has set (250 in our case). Another important digital signal processing is also done within the FPGA: the real IF-band signal becomes complex base-band signal two I/Q quadrature components. This also explains why the data rate after decimation is called 'quadrature rate'. Of course, the real story behind is much more complicated, we will leave the details to subsequent chapters. Finally, the complex base-band signal is sent to the software module in the computer via the USB 2.0 cable. A complex number is represented using a **real/imag** pair, which actually requires two real values.

OK, let's look at the code. *usrp* is the module we've imported at the beginning. The *usrp* module is located at:

/usr/local/lib/python2.4/site-packages/gnuradio/usrp.py

It tells us the module *usrp* is a wrapper for the USRP sinks (transmitting) and sources (receiving). When a sink or a source is instantiated, the *usrp* module first probes the USB port to locate the requested board number, then use the appropriate version specific sink or source. `source_c` is a function defined in *usrp* module. It returns a class object that represents the data source. The suffix `_c` means the data type of the signal is '**complex**', because the signal coming into the computer is complex (actually in real/imag pair). In contrast, we also have `source_s` method in the *usrp* module, which is designed for 16-bit short integer data type.

In this example, `source_c` takes two arguments. '0', specifies which USRP board is going to be opened. Just set it to zero if we work with only one USRP board. The second parameter tells the decimation rate to the USRP board. `set_rx_freq` and `set_pga` are two methods of the source '**src**'. `set_rx_freq` tells the USRP board the IF frequency. Just now we have mentioned the USRP board processes the real IF-band signal, into complex base-band signal with two I/Q quadrature components. To do this, the USRP board requires the knowledge of the IF frequency. **pga** is short for 'Programmable Gain Amplifier'. We can set its value (in db) using the `set_pga` method, which is 20db in our case.

Where are these methods defined? At this level, Python is insufficient to explain all these stuffs. Actually all these methods are implemented using C++. The **SWIG** provides the interfaces between C++ and Python, so that we can call these functions directly in Python, without worrying about the implementation details in C++. **Boost**, a smart pointer system, is also used here to facilitate the interaction between C++ and Python. So the story seems to be rather complicated. We skip the implementation details at this point. Let's just use these methods happily in Python!

An important document about USRP generated using Doxygen is located at:

/usr/local/share/doc/usrp-x.xcvs/html

We can look up all the methods provided by USRP in this document. The top level interfaces to the USRP are `usrp_standard_rx` and `usrp_standard_tx`. Also we should take a look at their base classes, `usrp_basic_rx`, `usrp_basic_tx` and `usrp_basic`. There are many other methods, to control and interact with the USRP board. Feel relaxed if you are still worried about the interfaces between Python and C++, we will get back to them when we talk about how to use C++ to write blocks later.

4.4.3 The big signal processing 'gut'

There is a long story behind

```
guts = blks.wfm_rcv (self, quad_rate, audio_decimation)
```

'**guts**' is the central processing block of this FM receiver. All signal processing blocks, such as deemphasizing, noncoherent demodulation are glued together in this 'gut'. This more interesting story can be found at

/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/wfm_rcv.py

The detailed FM receiving techniques will be discussed in Tutorial 7. Now let's just give it a shot from a high level point of view. `wfm_rcv` is a class defined in the module *blksimpl*. Its base class is `hier_block`, defined in the module `gr.hier_block`. `gr.hier_block` can be thought as a sub-graph, containing several signal processing blocks, which is used as a single sophisticated block in another bigger graph. In this statement, we create an object 'guts' as the instantiation of the class `wfm_rcv`. All real signal processing is done within this big block.

4.4.4 The signal sink

Finally, we will play the demodulated FM signal using the sound card. So the audio device is the signal sink in this example:

```
        # sound card as final sink
        audio_sink = audio.sink (int (audio_rate))
```

sink is a global function defined in the module *audio*. It returns an object as the signal sink block. `audio_rate` is a parameter describing the the data rate of the signal entering the sound card, which is 32kHz in our example.

4.4.5 Gluing them together

The next two lines finally complete our signal flow graph

```
        # now wire it all together
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))
```

Just now we have talked about the family of the flow graph classes, to which the new class `wfm_rx_graph` belong. All those flow graph classes are derived from the 'root' class `gr.basic_flow_graph`. The **connect** method is defined in `gr.basic_flow_graph`. This method is designed for the flow graph to bind all the blocks together. We will investigate more on flow graph classes and their methods in Tutorial 6.

The signal flow graph is done at this point!

5 Conclusion

OK! Let's stop here and have a rest for a while. The rest of the code requires some advanced knowledge. We will add GUI support to the FM receiver, which is cool and attractive. It is built upon *gr-wxgui*, which is based on wxPython and FFTW. We haven't talked about how to receive arguments from the user screen and how to start the flow graph. Some arguments passing among classes and functions may also seem to be confusing at the point. All these more advanced materials will be covered again in Tutorial 8.

This article is focusing on the basic syntax of Python and how Python plays its role in GNU Radio. Some software radio concepts and signal processing techniques are also mentioned along the way. I hope after reading this article, you can have a rough sense about how to program in GNU Radio.

APPENDIX A: The source code

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

from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math

from gnuradio.wxgui import stdgui, fftsink
import wx


class wfm_rx_graph (stdgui.gui_flow_graph):
    def __init__(self,frame,panel,vbox,argv):
        stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

        IF_freq = parseargs(argv[1:])
        adc_rate = 64e6

        decim = 250
        quad_rate = adc_rate / decim               # 256 kHz
        audio_decimation = 8
        audio_rate = quad_rate / audio_decimation  # 32 kHz

        # usrp is data source
        src = usrp.source_c (0, decim)
        src.set_rx_freq (0, IF_freq)
        src.set_pga(0,20)

        guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

        # sound card as final sink
        audio_sink = audio.sink (int (audio_rate))

        # now wire it all together
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))

        if 1:
            pre_demod, fft_win1 = \
                       fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                                512, quad_rate)
            self.connect (src, pre_demod)
            vbox.Add (fft_win1, 1, wx.EXPAND)

        if 1:
            post_deemph, fft_win3 = \
                         fftsink.make_fft_sink_f (self, panel, "With Deemph",
                                                  512, quad_rate, -60, 20)
            self.connect (guts.deemph, post_deemph)
            vbox.Add (fft_win3, 1, wx.EXPAND)

        if 1:
            post_filt, fft_win4 = \
                       fftsink.make_fft_sink_f (self, panel, "Post Filter",
                                                 512, audio_rate, -60, 20)
            self.connect (guts.audio_filter, post_filt)
            vbox.Add (fft_win4, 1, wx.EXPAND)


def parseargs (args):
    nargs = len (args)
    if nargs == 1:
        freq1 = float (args[0]) * 1e6
    else:
        sys.stderr.write ('usage: wfm_rcv freq1\n')
        sys.exit (1)

    return freq1 - 128e6

if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

References

[1] **Python on-line tutorials**, http://www.python.org/doc/current/tut/

[2] Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

Tutorial 5: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line - Part I

Dawei Shen[1]

May 27, 2005

Abstract

Python plays a key role in GNU Radio programming. GNU Radio provides a framework for building software radios. The signal processing applications - are built using a combination of Python code for high level organization, policy, GUI and other non performance-critical functions, while performance critical signal processing blocks are written in C++. From the Python point of view, GNU Radio provides a data flow abstraction. This article is focused on the Python level, introducing the basic usage of Python and how Python is used in GNU Radio to glue all the signal processing blocks and control the flow of the digital data. A popular and classical example: the implementation of an FM receiver with GUI, is used here. The code is analyzed line by line. The basic grammar of Python and the concept of software radio are introduced in parallel. So this article can be used as both a short Python tutorial and a software radio guidance. How to write the blocks using C++ and other advanced topics will be covered in subsequent chapters.

Contents

1 Overview

GNU Radio's software is organized using a two-tier structure. All the performance-critical signal processing blocks are implemented in C++, while the higher-level organizing, connecting and gluing are done using Python. Many frequently used signal processing blocks have been implemented well and provided to us as parts of the GNU Radio software.

This structure has some similarity with the OSI 7-layer data network model. Lower layer provides service to the higher layer, while the higher layer doesn't care about the implementation details carried on in lower layers, but necessary interfaces and function calls. In GNU Radio, this layer transparency exists in a similar way. From the Python's point of view, what it does is just to select necessary signal sources, sinks and processing blocks, set correct parameters, then connect them together to form a complete application. In fact, all these sources, sinks and blocks are implemented as classes in C++. The parameter setting, connecting operations correspond to some sophisticated functions or class methods in C++. However, Python can't see how sedulously C++ has been working. A piece of lengthy, complicated and powerful C++ code is nothing but an interface to Python.

As a result, no matter how complicated the application is, the Python code is almost always short and neat. The real heavy load is thrown to C++. A thumb of rule should be kept in mind: for any application, what we need to do at the Python level, is always just to draw a diagram showing the signal flow from the source to the sink in your mind, then use the `nice pen' - Python, to find them and connect them all together, sometimes with the graphical user interfaces (GUI) support.

Obviously Python is crucial in learning GNU Radio. Python is a powerful and flexible programming language, which itself is a long story. But if you have enough C/C++ background, it's just a piece of cake. Considering the fact that Python has some special characteristics when applied to GNU Radio and that some of its cool fancy features may not be necessary in GNU Radio, the article aims at combining the Python programming techniques and software radio concepts together. I am a believer that the best way to grasp the essence of some new knowledge is to go through an example rather than learning the syntax or semantics dryly. So our discussion will surround a popular and classical example: the implementation of an FM receiver with graphical user interfaces. The code will be analyzed line by line and the Python programming, signal processing techniques, software radio concepts and some hardware configuration will be talked about along the way.

This example implements an FM receiver with graphical user interface. The FM signal from the air is received by the USRP board then gets processed in the USRP board and in the computer. Finally the demodulated signal is played using the sound card. No fancy antenna is required. You can hear very high quality FM signal by just inserting a copper wire to the basic RX daughter board. The code can be found at '`gnuradio-examples/python/usrp/wfm_rcv_gui.py`'. Please refer to appendix A to make sure we have the same version of the code. We plan to use two articles to cover the whole materials. This article is the first half, focusing on the basics of Python and GNU Radio. Some advanced topics, such as the GUI tools (wxPython) and the usage of some Python built-in packages, are left to Tutorial 8.

2 The first line

If you have read the code of other examples, you can find the first line of these programs is almost always

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

The Python scripts can be made directly executable if we put this line at the beginning of the script and giving the file an executable mode. The '#!' must be the first two characters of the file. The script is given the executable mode by using the '**chmod**' command:

```
$ chmod +x wfm_rcv_gui.py
```

Now the script `wfm_rcv_gui.py` becomes executable. You can run this program in the shell using

```
$ ./wfm_rcv_gui.py arguments
```

the Python interpreter will be invoked and the code in this script will be executed line by line orderly. Python is an interpreted language, like Matlab script. No compilation and linking is necessary.

There are several ways to invoke the Python interpreter: you can use

```
$ python ./wfm_rcv_gui.py arguments
```

without the need to give the script the executable mode.

You can also use the interactive mode, by just typing the command in the shell:

```
$ python
```

then the Python interpreter environment will be invoked and you could input your code line by line. However, this is obviously inconvenient. We seldom use the interpreter interactively, unless we write some throw-away programs, test functions or use it as a desk calculator. Most of the time, packing codes in a **.py** file and make the script self-executable is more convenient to us.

3 Importing necessary modules

Next, we see a lot of importing stuffs:

```
from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
```

```
    from gnuradio import blks
    from gnuradio.eng_option import eng_option
    from optparse import OptionParser
    import sys
    import math
    from gnuradio.wxgui import stdgui, fftsink
    import wx
```

Understanding these statements requires the knowledge of '**module**' and '**package**' concepts in Python. The best way to learn them is to go through the Chapter 6 of the Python tutorials.

Here is a brief introduction. If we quit the Python interpreter and enter it again, all the functions and variables we have defined are lost. Therefore, we wish to write a somewhat longer program and save it as a **script**, containing function and variable definitions, and maybe some executive statements. This script can be used as the input of the Python interpreter. We may also want to use a fancy function we've written in several programs without copying its definitions to each program.

To support this, Python provides a **module**/**package** organization system. A **module** is a file containing Python definitions and statements, with the suffix '**.py**'. Within a module, the module's name (as a string) is available as the value of the global variable '`__name__`'. Definitions in a module can be imported into other modules or into the top-level module. A **package** is a collection of modules that have similar functions, which are often put in the same directory. The '`__init__.py`' files are required to make Python treat the directories as packages. A package could contain both modules and sub-packages (can contain sub-sub-packages). We use 'dotted module names' to structure Python's module namespace. For example, the module name A.B designates a submodule named 'B' in a package named 'A'.

A module can contain executable statements as well as function definitions. The statements are executed only the first time the module is imported somewhere and so that the module is initialized. Each module has its own private symbol table, which is used as the global symbol table by all functions defined within that module. The author of a module can use the global variables in the module without worrying about accidental clashes with the module user's global variables. As a module user, we can access a module's function and global variables using `**modname.itemname**'. Here the `**itemname**' can either be a function or a variable.

Modules can import other modules using `**import**' command. It is customary to place all `**import**' statements at the beginning of a module. Note that the **import** operation is quite flexible. We can import a package, a module, or just a definition within a module. When we try to import a module from a package, we can either use `**import packageA.moduleB**', or `**from package A import module B**'. When using `**from package import item**', the '**item**' can be either a module/sub-package of the package, or some other names defined in the package, like functions, classes or variables.

It's worth taking a while to introduce the modules used in this example amply, because these modules or packages will be frequently encountered in GNU Radio. The top-level package of GNU Radio is '*gnuradio*', which includes all GNU Radio related modules. It is located at

```
/usr/local/lib/python2.4/site-packages
```

By default, this directory is not included in the Python's search path, we need to export the path to the environment variable '**PATHONPATH**'. So we usually add the following line to the users' `.bash_profile` file:

```
$export PATHONPATH=/usr/local/lib/python2.4/site-packages
```

to make sure the Python interpreter could find the *gnuradio* package.

gr is an important sub-package of *gnuradio*, which is the core of the GNU Radio software. The type of '**flow graph**' classes is defined in *gr* and it plays a key role in scheduling the signal flow. *eng_notation* is a module designed for engineers' notation convenience, in which many words and characters are endowed with new constant values according to the engineering convention. The module *audio* provides the interfaces to access the sound card, while *usrp* provides the interfaces to control the USRP board. *audio* and *usrp* are often used as the signal source and sink. We will see the details about them later in this article. *blks* is a sub-package, which is almost an empty folder if you check its directory. It actually transfers all its tasks to another sub-package *blksimpl* in *gnuradio*, as described in the `__init__.py` file. *blksimpl* provides the implementation of several useful applications, such as FM receiver, GMSK, etc. For this example, the real signal processing part of the FM receiver is performed in this package.

Look at the next line, which is more interesting:

```
from gnuradio.eng_option import eng_option
```

This is exactly what we mentioned just now, we can either import a complete module/sub-package, or, just a function, class or variable definition from this module. In this case, *eng_option* is a class defined in the module *gnuradio.eng_option*. We don't need the whole module to be imported, but just a single class definition. *gnuradio.eng_option* module does nothing but adding support for engineering notation to *optparse.OptionParser*.

This line appears to have a similar format:

```
from gnuradio.wxgui import stdgui, fftsink
```

But the meaning is a little bit different, *gnuradio.wxgui* is a sub-package, not a module, while *stdgui* and *fftsink* are two modules in this sub-package. It's not necessary to import the whole sub-package, so we just import what we want explicitly. *gnuradio.wxgui* provides visualization tools for GNU Radio, which is constructed based on wxPython. The importing operations in Python provide us great flexibility and convenience.

Finally, *optparse*, *math*, *sys*, *wx* are all Python or wxPython's built-in modules or sub-packages, which are not part of the GNU Radio.

At this point, let me emphasize again, these modules imported above may contain executable statements as well as the function or class definitions. The statements will be executed immediately after the modules are imported. After importing the modules and packages, a lot of variables, classes and modules defined in them have been initialized. So don't think nothing has been done. Actually a lot of work behind the stage has been carried on. Many guys are waiting for your order in the workspace.

OK! So far we have taken a quick look at the most frequently used modules in GNU Radio and seen how they are organized together. Maybe it sounds too abstract and you are still confused about their usage. Never mind, we will see them soon.

4 The story in the class wfm_rx_graph

Familiarizing with object oriented programming (OOP) is important for understanding this section. OOP itself is a long story, which is obviously not what we are focusing on. But we will talk about some OOP concepts along the way. Some digital signal processing techniques will also be discussed as a reminder if we meet the corresponding codes.

4.1 Class definition

In this example, a large part of the code is the definition of a class '`wfm_rx_graph`'. The statement

```
class wfm_rx_graph (stdgui.gui_flow_graph):
```

defines a new class '`wfm_rx_graph`', which is inherited (derived) from the base class, or the so called 'father class' `--`'`gui_flow_graph`'. The father class `gui_flow_graph` is defined in the module *stdgui* we have just imported from *gnuradio*. By the rules of the namespace, we use `stdgui.gui_flow_graph` to refer to it.

4.2 The family of 'FLOW GRAPH' classes

Here an important category of classes, which play a key role in GNU Radio should receive particular attention: the 'flow graph' classes. There are a series of 'GRAPH' related classes defined in GNU Radio. We can find that `stdgui.gui_flow_graph` is derived from `gr.flow_graph`, which is defined in the sub-package *gr*. Further, `gr.flow_graph` is derived from the 'root' class `gr.basic_flow_class`. In GNU Radio, there are also many other classes derived from `gr.basic_flow_graph`. This big 'GRAPH family' makes GNU Radio programming neat and simple, also makes the scheduling of the signal processing clear and straightforward.

What do these 'graphs' do? Suppose you are trying to design a circuit using some commercial software such as Pspice. Probably you will first open a schematic, or a 'canvas', then you put all necessary circuit parts, such as a resistor, an amplifier or some DC powers on this canvas. Finally, you draw lines among these parts to connect them together and complete the circuit design. This scenario applies perfectly into GNU Radio. A **graph** is just like the schematic or the canvas. The circuit parts are replaced by signal sources, sinks and the signal processing blocks in GNU Radio. Finally, those 'wires' correspond to the '**connect**' method of the graph class, which is in charge of gluing these blocks together. Definitely, sometimes, an integrated circuit can serve as a part, the so called sub-circuit, in another schematic. That's also true in GNU Radio, a sub-graph can be used as a whole block in another graph.

In our example, `wfm_rx_graph` is such a graph class belonging to this family, with GUI support. Later we will see it glues the necessary blocks in FM receiver together using the method '**connect**'.

4.3 The initialization function: __init__

Then we implement the method (or function) '`__init__`' of the class `wfm_rx_graph`. The syntax for defining a new method is

```
def funcname(arg1 arg2 ...)
```

`__init__` is an important method for any class. After defining the class, we may use the class to instantiate an instance. This special method `__init__` is used to create an object in a known initial state. Class instantiation automatically invokes `__init__` for the newly created class instance. Actually in this example, `__init__` is the only method defined in the class `wfm_rx_graph`.

One important feature of Python is worth mentioning before we talk about the details in the function `__init__`. We notice that in this piece of code, there is no explicit signs for where a definition of a class or a function starts and ends. Usually in other programming languages such as C++ or Pascal, we use '**begin**' and '**end**' pair, or a pair of '{' and '}' explicitly to denote the two ends of a group of statements. However, in Python, this is no longer the case. There is **NO** such signs. In Python, statement grouping is done by **indentation** instead of beginning and ending brackets. So be careful about your editing and layout of the code when you write programs using Python.

Now let's see what's going on in the function `__init__`.

```
def __init__(self,frame,panel,vbox,argv):
```

declares the initialization method `__init__` with four arguments. Conventionally, the first argument of all methods are often called **self**. This is nothing more than a convention: the name **self** has absolutely no special meaning to Python. However, methods may call other methods by using method attributes of the **self** argument, such as '`self.connect()`' we will meet later.

The first thing `__init__` does, is to call the initialization method of `stdgui.gui_flow_graph`, its 'father class' , with exactly the same four arguments.

```
stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,
argv)
```

You may like to take a look at `stdgui.gui_flow_graph`'s initialization method. Since `wfm_rx_graph` is derived from it, we can safely think of `wfm_rx_graph` as a 'special' `gui_flow_graph`. So it's natural that this 'son class' should do something to make himself look like his father first, then do something fancy to make himself a different guy.

4.4 Constructing the graph with source, sink and signal processing blocks

4.4.1 Defining constants

From the next line, we kind of start to see the real signals coming in, which is less boring

```
IF_freq = parseargs(argv[1:])
```

sets the IF frequency from the return value of the function **parseargs**.

What does IF frequency stand for? IF is short for 'intermediate frequency'. Roughly speaking, it's the center frequency of the frequency band we are interested in. We move the `real' frequency band, the so called RF band, which is usually very high, to some intermediate frequency band (IF) where the ADC can work obeying the Nyquist theorem. This isn't the key point in this article. I hope you have already got enough background in communications and DSP at this point.

The function **parseargs** is defined in this example later, right after the definition of the class `wfm_rx_graph`. It accepts the user's input arguments when the program is executed in shell. Let's talk about it later.

Besides, you may have noticed that Python doesn't require variable or argument declarations. This is totally different with the `declare before use' concept in C.

```
adc_rate = 64e6
```

This line defines the sampling frequency of the AD converter, which should be set to 64MHz for the USRP users. According to Nyquist theorem, the maximum frequency component of the interested signal should be less than 32MHz in order not to loose spectrum information after sampling.

```
decim = 250
quad_rate = adc_rate / decim              # 256 kHz
```

Decimation is a concept in DSP world. After sampling the analog signal, we get a digital signal with very high data rate, which is a heavy burden for the CPU and storage. Usually, we can down-sample the digital sequence (decimation) without losing the spectrum information. In this example, the decimation rate is chosen to be 250 so that the resulting data rate is 256K samples per second, which is quite reasonable and acceptable for our CPU speed. `quad_rate` represents for **quadrature data rate**. The reason why it is called quadrature rate will be explained later.

``# 256 kHz`' is nothing but a piece of comment. In Python, a comment starts after the symbol `#'. All statements after `#' in each line will be ignored by Python interpreter.

```
    audio_decimation = 8
    audio_rate = quad_rate / audio_decimation              # 32 kHz
```

After processing the digital FM signal, we wish to play the signal using the sound card of the computer. However, the data rate that a sound card can adopt is rather limited. 256kHz is usually too high and more than necessary. So we need to further decimate the data rate. 32kHz is a common choice for most sound cards.

4.4.2 The signal source

The following several lines provide a very high level abstraction for the signal processing procedure, which basically includes three components: the signal source, the signal sink, and a series of signal processing blocks. This example gives those signal processing blocks a very nice name: **guts**.

The signal source for the FM receiver is the USRP board in this example

```
    # usrp is data source
    src = usrp.source_c (0, decim)
    src.set_rx_freq (0, IF_freq)
    src.set_pga(0,20)
```

The USRP board receives the analog FM signal from the air via the RX daughter board and samples the signal using the AD converter with a sampling rate 64MHz. The resulting digital sequence then goes into the FPGA chip equipped on the USRP board. It is down-sampled there according to the decimation rate that the user has set (250 in our case). Another important digital signal processing is also done within the FPGA: the real IF-band signal becomes complex base-band signal two I/Q quadrature components. This also explains why the data rate after decimation is called `quadrature rate'. Of course, the real story behind is much more complicated, we will leave the details to subsequent chapters. Finally, the complex base-band signal is sent to the software module in the computer via the USB 2.0 cable. A complex number is represented using a **real/imag** pair, which actually requires two real values.

OK, let's look at the code. *usrp* is the module we've imported at the beginning. The *usrp* module is located at:

```
    /usr/local/lib/python2.4/site-packages/gnuradio/usrp.py
```

It tells us the module *usrp* is a wrapper for the USRP sinks (transmitting) and sources (receiving). When a sink or a source is instantiated, the *usrp* module first probes the USB port to locate the requested board number, then use the appropriate version specific sink or source. `source_c` is a function defined in *usrp* module. It returns a class object that represents the data source. The suffix `_c` means the data type of the signal is `**complex**', because the signal coming into the computer is complex (actually in real/imag pair). In contrast, we also have `source_s` method in the *usrp* module, which is designed for 16-bit short integer data type.

In this example, `source_c` takes two arguments. `0', specifies which USRP board is going to be opened. Just set it to zero if we work with only one USRP board. The second parameter tells the decimation rate to the USRP board. `set_rx_freq` and `set_pga` are two methods of the source `**src**'. `set_rx_freq` tells the USRP board the IF frequency. Just now we have mentioned the USRP board processes the real IF-band signal, into complex base-band signal with two I/Q quadrature components. To do this, the USRP board requires the knowledge of the IF frequency. **pga** is short for `Programmable Gain Amplifier'. We can set its value (in db) using the `set_pga` method, which is 20db in our case.

Where are these methods defined? At this level, Python is insufficient to explain all these stuffs. Actually all these methods are implemented using C++. The **SWIG** provides the interfaces between C++ and Python, so that we can call these functions directly in Python, without worrying about the implementation details in C++. **Boost**, a smart pointer system, is also used here to facilitate the interaction between C++ and Python. So the story seems to be rather complicated. We skip the implementation details at this point. Let's just use these methods happily in Python!

An important document about USRP generated using Doxygen is located at:

```
/usr/local/share/doc/usrp-x.xcvs/html
```

We can look up all the methods provided by USRP in this document. The top level interfaces to the USRP are `usrp_standard_rx` and `usrp_standard_tx`. Also we should take a look at their base classes, `usrp_basic_rx`, `usrp_basic_tx` and `usrp_basic`. There are many other methods, to control and interact with the USRP board. Feel relaxed if you are still worried about the interfaces between Python and C++, we will get back to them when we talk about how to use C++ to write blocks later.

4.4.3 The big signal processing 'gut'

There is a long story behind

```
guts = blks.wfm_rcv (self, quad_rate, audio_decimation)
```

`**guts**' is the central processing block of this FM receiver. All signal processing blocks, such as deemphasizing, noncoherent demodulation are glued together in this `gut'. This more interesting story can be found at

```
/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/
wfm_rcv.py
```

The detailed FM receiving techniques will be discussed in Tutorial 7. Now let's just give it a shot from a high level point of view. `wfm_rcv` is a class defined in the module *blksimpl*. Its base class is `hier_block`, defined in the module `gr.hier_block`. `gr.hier_block` can be thought as a sub-graph, containing several signal processing blocks, which is used as a single sophisticated block in another bigger graph. In this statement, we create an object `guts' as the instantiation of the class `wfm_rcv`. All real signal processing is done within this big block.

4.4.4 The signal sink

Finally, we will play the demodulated FM signal using the sound card. So the audio device is the signal sink in this example:

```
# sound card as final sink
audio_sink = audio.sink (int (audio_rate))
```

sink is a global function defined in the module *audio*. It returns an object as the signal sink block. `audio_rate` is a parameter describing the the data rate of the signal entering the sound card, which is 32kHz in our example.

4.4.5 Gluing them together

The next two lines finally complete our signal flow graph

```
# now wire it all together
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

Just now we have talked about the family of the flow graph classes, to which the new class `wfm_rx_graph` belong. All those flow graph classes are derived from the `root' class `gr.basic_flow_graph`. The **connect** method is defined in `gr.basic_flow_graph`. This method is designed for the flow graph to bind all the blocks together. We will investigate more on flow graph classes and their methods in Tutorial 6.

The signal flow graph is done at this point!

5 Conclusion

OK! Let's stop here and have a rest for a while. The rest of the code requires some advanced knowledge. We will add GUI support to the FM receiver, which is cool and attractive. It is built upon *gr-wxgui*, which is based on wxPython and FFTW. We haven't talked about how to receive arguments from the user screen and how to start the flow graph. Some arguments passing among classes and functions may also seem to be confusing at the point. All these more advanced materials will be covered again in Tutorial 8.

This article is focusing on the basic syntax of Python and how Python plays its role in GNU Radio. Some software radio concepts and signal processing techniques are also mentioned along the way. I hope after reading this article, you can have a rough sense about how to program in GNU Radio.

APPENDIX A: The source code

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

from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math

from gnuradio.wxgui import stdgui, fftsink
import wx


class wfm_rx_graph (stdgui.gui_flow_graph):
    def __init__(self,frame,panel,vbox,argv):
        stdgui.gui_flow_graph.__init__ (self,frame,panel,vbox,argv)

        IF_freq = parseargs(argv[1:])
        adc_rate = 64e6

        decim = 250
        quad_rate = adc_rate / decim               # 256 kHz
        audio_decimation = 8
        audio_rate = quad_rate / audio_decimation  # 32 kHz

        # usrp is data source
        src = usrp.source_c (0, decim)
        src.set_rx_freq (0, IF_freq)
        src.set_pga(0,20)

        guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

        # sound card as final sink
        audio_sink = audio.sink (int (audio_rate))
        # now wire it all together
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))

        if 1:
            pre_demod, fft_win1 = \
                       fftsink.make_fft_sink_c (self, panel, "Pre-
Demodulation",
                                                512, quad_rate)
            self.connect (src, pre_demod)
            vbox.Add (fft_win1, 1, wx.EXPAND)

        if 1:
            post_deemph, fft_win3 = \
                         fftsink.
make_fft_sink_f (self, panel, "With Deemph",
                                                  512, quad_rate, -
60, 20)
            self.connect (guts.deemph, post_deemph)
            vbox.Add (fft_win3, 1, wx.EXPAND)

        if 1:
            post_filt, fft_win4 = \
                       fftsink.
make_fft_sink_f (self, panel, "Post Filter",
                                                512, audio_rate, -
60, 20)
            self.connect (guts.audio_filter, post_filt)
            vbox.Add (fft_win4, 1, wx.EXPAND)


def parseargs (args):
    nargs = len (args)
    if nargs == 1:
        freq1 = float (args[0]) * 1e6
    else:
        sys.stderr.write ('usage: wfm_rcv freq1\n')
        sys.exit (1)

    return freq1 - 128e6

if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

References

[1]

Python on-line tutorials, http://www.python.org/doc/current/tut/

[2]

Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

---

Footnotes:

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

---

Tutorial 6: Graph, Blocks & Connecting

Dawei Shen[*]

June 7, 2005

Abstract

In tutorial 5, we have seen the family of flow graph classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a flow graph accommodates several signal sources, sinks and signal processing blocks then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

1 Overview

In tutorial 5, we have seen the family of ‘flow graph’ classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a ‘flow graph’ accommodates several signal sources, sinks and signal processing blocks, then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

Another example, ‘dial tone’, is chosen for illustration purpose. This is a very simple ‘Hello World!’ style example. But it should be enough to demonstrate the strength and beauty of the graph mechanism in GNU Radio. In this example, we simply generate two sine waves of different frequency and play the tones through the sound card. Only the signal source and sink are involved, without real signal processing.

The source code of this example is located at: ’`gnuradio-examples/python/audio/dial_tone.py`’. Please refer to appendix A to make sure we have the same version of the code. To run this example, no USRP board is required, but the sound card equipped on your computer. Simply input the command `$./dial_tone.py` in your shell, you can hear clear tones from your speaker.

A few Python programming tips will also be reminded along the way.

2 Defining a function build_graph()

Two modules *gr* and *audio* are imported first. Then a function `build_graph()`, with no arguments, is defined:

```
def build_graph ():
```

`build_graph ()` is a global function within the module `dial_tone.py`, slightly different with the `__init__()` we talked about last time, which is a method belonging to the class `wfm_rx_graph`.

Next two constants are defined. Just a reminder: don’t forget the indentation!

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

```
    sampling_freq = 32000
    ampl = 0.1
```

We have met `sampling_freq` before when we talked about the audio decimation in last tutorial. It is the data rate of the digital signal entering the sound card. For most sound cards we use, 32kHz is a fairly good choice. The sampling rate is a parameter for the signal source `gr.sig_source_f` we will see later. Another parameter is the amplitude of the sine waves, `ampl`. We set it to 0.1v in this example.

3 Creating a graph

The graph comes!

```
fg = gr.flow_graph ()
```

`flow_graph` is a class defined in the module `gr.flow_graph.py`.

Here is a trick. Why could we use `gr.flow_graph` directly to refer to this class, not `gr.flow_graph.flow_graph`? The answer is in the initialization file '`__init__.py`' of the package *gr*. Let's look at its content:

```
    from gnuradio_swig_python import *
    from basic_flow_graph import *
    from flow_graph import *
    from exceptions import *
    from hier_block import *
```

We know a directory must contain a `__init__.py` file to let Python treat it as a *package*. The statements in `__init__.py` will be executed first immediately after the package is imported somewhere. In this case, when *gr* is imported, all the definitions in `flow_graph` are imported to the package *gr*'s global symbol table. Here '*' represents for 'everything'. So we can directly access to the classes or functions defined in `flow_graph.py` using `gr.item` rather than `gr.flow_graph.item`.

`flow_graph` is derived from its 'father class' `basic_flow_graph`, the root of all 'GRAPH' related classes. `basic_flow_graph` describes the connections between blocks conceptually. Let's take a look at its source file located at:

```
/usr/local/python2.4/site-packages/gnuradio/gr/basic_flow_graph.py
```

By common sense, a graph should consist of vertices as well as the edges connecting them. These two basic elements are defined as `endpoint` class and `edge` class in `basic_flow_graph.py`.

Part of the definition for `endpoint` is

```
class endpoint (object):
      __slots__ = ['block', 'port']
      def __init__ (self, block, port):
          self.block = block
          self.port = port
```

In Python, '`__slot__`' is a special variable of a class. By default, all instances of classes have a 'dictionary' for attribute storage: the variable '`__dict__`', the data type of which is '**Dictionary**'. This mechanism wastes space for objects having very few variables. The space consumption can become acute when creating large numbers of instances. The `__slots__` declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because `__dict__` is not created for each instance.

So from `__slot__`, we can see the `endpoint` class has two attributes: `block` and `port`. This 2-tuple can fully specify an end point (vertice) in a graph. A `block` can be a source, a sink, or a signal process block. One `block` may have multiple ports, corresponding to different vertices in the graph.

This is part of the definition for the class `edge`:

```
class edge (object):
      __slots__ = ['src', 'dst']
      def __init__ (self, src_endpoint, dst_endpoint):
          self.src = src_endpoint
          self.dst = dst_endpoint
```

An `edge` provides a directed connection between two end points. Obviously, `edge` has two attributes: `src` and `dst`, indicating where the signal flow starts and ends. The data type for `src` and `dst` is just the class '`endpoint`' we have defined.

Based on `endpoint` and `edge`, we define the class `basic_flow_graph`. Part of its definition is:

```
class basic_flow_graph (object):
      '''basic_flow_graph -- describe connections between blocks'''
      __slots__ = ['edge_list']
      def __init__ (self):
          self.edge_list = []
```

`basic_flow_graph` has only one attribute '`edge_list`', which is a **List** saving all the edges in the graph. It is initialized to be empty when a flow graph instance is created. The 'list' will be modified only if we call the methods defined in the class `basic_flow_graph`, such as '`connect`' or '`disconnect`'. We will talk about these methods later.

The class `flow_graph` used in our example is derived from `basic_flow_graph`. It is defined in

```
/usr/local/python2.4/site-packages/gnuradio/gr/flow_graph.py
```

This is part of its definition:

```
class flow_graph (basic_flow_graph):
      """add physical connection info to simple_flow_graph
      """
      __slots__ = ['blocks', 'scheduler']

      def __init__ (self):
          basic_flow_graph.__init__ (self);
          self.blocks = None
          self.scheduler = None
```

The comment reveals everything. Compared with its 'father class' `basic_flow_graph`, `flow_graph` adds the physical connection to it. `basic_flow_graph` only defines a flow graph 'abstractly', preserving the information about what kind of 'end points' have been used in the graph and which ones of them are connected by the directed 'edges'. You can think of it as a diagram drawn on your draft paper, nothing in the real world occurs. Of course neither can you run the graph. In `flow_graph`, we connect the blocks 'physically' , i.e., all blocks' inputs are connected to their upstream buffers – real buffers in the computer memory. To understand the 'physical connection' more thoroughly, we can look at the following methods defined in the class `flow_graph`:

```
   def _setup_connections (self):
       """given the basic flow graph, setup all the physical connections"""
       self.validate ()
       self.blocks = self.all_blocks ()
       self._assign_details ()
       self._assign_buffers ()
       self._connect_inputs ()

   def _assign_details (self):
       ...
   def _assign_buffers (self):
```

```
            """determine the buffer sizes to use, allocate them and attach to detail"""
            ...
        def _connect_inputs (self):
            """connect all block inputs to appropriate upstream buffers"""
            ...
```

When a graph instance calls its `start()` method to run the program, the graph will first call its `_setup_connections()` method to setup all the physical connections. It actually calls three methods: `_assign_details()`, `_assign_buffers()` and `_connect_inputs()` sequentially. The three methods finish the real connection job on a 'physical' level. The required buffer size is calculated and then allocated. Finally all the blocks' input are connected to appropriate upstream buffers. The implementation details in these methods are overdetailed for us and seldom needed when we do programming with GNU Radio. We never call these methods explicitly when we play with the graph. However, I believe knowing some of the story hidden behind could help us obtain a better understanding about the operating mechanism of GNU Radio.

`flow_graph` adds two more attributes: '`blocks`' and '`scheduler`'. `blocks` is a **List** variable, keeping the list of all blocks existing in the graph. It uses the method `all_blocks()` defined in `basic_flow_graph` to get the list. Two endpoints with the same block but different ports will be counted only once. `scheduler` is an important variable controlling the running of the flow graph. Its data type is the class `scheduler`, defined in the module *gr.scheduler*. `gr.scheduler` uses Python's built-in module `threading`, to control the 'starting', 'stopping' or 'waiting' operations of the graph.

The most distinguishing parts that `flow_graph` adds to the `basic_flow_graph` are the methods that control the running of the flow graph, such as `start()`, `stop()`, `wait()` `run()`, etc. As indicated by their names, these methods are designed to actually drive the signal flows to move on or stop in the graph. `scheduler` plays a key role in these methods.

We can imagine that for most of the time when we want to create a graph instance or derive a graph class, we should use `flow_graph`, rather than `basic_flow_graph`.

OK! We have got so many stories from this single line: `fg = gr.flow_graph ()`. I hope you haven't forgotten, we have created a graph! It's an instance of the class `flow_graph`.

One final comment, in the introduction above, we mentioned several Python's new data types, such as '**List**' '**tuple**' and '**Dictionary**'. They are actually part of the reason why Python is so powerful, flexible and efficient. But unfortunately, we won't cover all these details in this tutorial. Please refer to the Python tutorial to gather more information about them.

4 Signal sources and sinks

4.1 The sources

Let's move on. Next we create two data sources. In last tutorial, we have seen one type of source: *usrp*. It takes the USRP board as the signal input. In this example, the source is much simpler: sine waves generated by the computer:

```
        src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
        src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
```

Strictly speaking, this signal source also belongs to the category of '**blocks**'. As mentioned before, in GNU Radio, all the blocks are implemented using C++. The **SWIG** provides the interfaces between Python and C++, so that in Python you can directly use the class definitions or functions implemented using C++. We will talk about how to write blocks using C++ and how to use **SWIG** to construct the interfaces in tutorial 10. That will be the best time to understand the whole story. At this point, let's forget those annoying tricks and just use those blocks safely in Python.

The best way to learn what blocks have been implemented and bundled in GNU Radio is to look at the documentation of GNU Radio generated using Doxygen. After the installation of GNU Radio, it should be located at:

/usr/local/share/doc/gnuradio-core-2.5cvs/html/index.html

It's also available on-line here.

Press the tag 'Class Hierarchy' and look for the class `gr_sig_source_f`. Once you find it, open the class reference for it, you can see the class hierarchy of `gr_sig_source_f`. The 'root' class is `gr_block`. All signal processing blocks are derived from `gr_block`. `gr_sync_block` is an important class derived from `gr_block`. It implements a 1:1 block with optional history and certain simplifications are made. Sources and sinks , such as `gr_sig_source_f`, are derived from `gr_sync_block`. The only thing different about them is that sources have no inputs and sinks have no outputs. All these blocks are started with the prefix '`gr_`' and are implemented using C++. There is some behind-the-scenes magic when we use **SWIG** to provide the interface between C++ and Python, so that we can access `gr_sig_source_f` from Python as `gr.sig_source_f()`. The suffix '`_f`' here, indicates the data type of the source, which is 'float' in our example. All these magics will be touched again when we talk about writing blocks in C++, so don't worry if you have any confusion at this point.

Anyway, the bottom line is, we have created two source instances returned by `gr.sig_source_f()`. They can be drawn into the 'graph' we created just now!

4.2 The sinks

We have seen the same sink before in the FM receiver: the audio device. We need to play the tones generated using the sound card.

```
dst = audio.sink (sampling_freq)
```

The *audio* module is located at:

/usr/local/lib/python2.4/site-packages/gnuradio/audio.py

The module *audio* is quite similar as *usrp*. It provides a wrapper for the audio sinks and sources. Again, there are some delicate magics behind the scene, so that you can create an audio sink instance in Python directly using `audio.sink()`. '`sink()`' and '`source()`' are actually functions defined in the module *audio*, which are connected to the OSS or ALSA supports for the sound cards, assuming you have installed *gr-audio-oss* or *gr-audio-alsa* modules. These two methods will return a class instance as the audio sink or source.

The story hidden from your view may be complicated, but constructing an audio sink in Python is really simple. We have done it! Let's forget those annoying details.

5 Connecting

Finally, we connect the blocks we have defined together to complete our flow graph:

```
fg.connect (src0, (dst, 0))
fg.connect (src1, (dst, 1))
```

The usage of the '`connect()`' method is quite straightforward. But there are some issues worth mentioning. Let's visit the definition of the class `basic_flow_graph` again. This is the definition for the method `connect()`:

```
def connect (self, *points):
    '''connect requires two or more arguments that can be coerced to
    endpoints. If more than two arguments are provided, they are
    connected together successively.
    '''
    if len (points) < 2:
```

```
                raise ValueError, ("connect requires at least two endpoints;
                %d provided." % (len (points),))
            for i in range (1, len (points)):
                self._connect (points[i-1], points[i])
```

The '`*`' before the argument `points` is a special feature of Python's function. It means the function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a **tuple**. Before the variable number of arguments, zero or more normal arguments may occur. So the `connect()` method can actually accept more than 2 arguments. If more than two blocks are provided to `connect()` as arguments, they will be connected one by one successively, on which the code is quite clear.

Another issue hidden behind is, when we connect the points, the points are first 'coerced' to be an 'endpoint' instance. It's necessary to recast the argument first to make it consistent with the class `endpoint`. Let's see the definition of the method `coerce_endpoint()`:

```
        def coerce_endpoint (x):
            if isinstance (x, endpoint):
                return x
            elif isinstance (x, types.TupleType) and len (x) == 2:
                return endpoint (x[0], x[1])
            elif hasattr (x, 'block'):          # assume it's a block
                return endpoint (x, 0)
            elif isinstance(x, hier_block.hier_block_base):
                return endpoint (x, 0)
            else:
                raise ValueError, "Not coercible to endpoint: %s" % (x,)
```

It's obvious that we don't need to provide an entire `endpoint` instance as the argument to the `connect()` method (in fact doing this will make coding messy). We can simply provide a 2-tuple as the arguments, i.e. (block name, port No.). The `coerce_endpoint()` method will convert it to an `endpoint` instance. Like in our example, we refer to the end points as `(dst, 0)` and `(dst, 1)`. If we only provide the block instance without the port number, the `coerce_endpoint()` method will add 0 as the default port number and return an `endpoint` instance, like the `src` in our example. It's equivalent to say `(src, 0)`. For blocks with only one port, it's convenient to directly use the block name as the argument for `connect()`.

Note that as mentioned above, at this point, the graph is only created and connected logically, not physically. We only have a diagram on the draft paper.

Finally, the global function `build_graph()` returns the the flow graph instance `fg` we have created.

6 Run the program

Now it's time to make the flow graph running.

```
        if __name__ == '__main__':
            fg = build_graph ()
            fg.start ()
            raw_input ('Press Enter to quit: ')
            fg.stop ()
```

What's the first line mean? Every module has an attribute '`__name__`', to indicate the name of the current module. Many modules contain a paragraph of code like this:

```
        if __name__ == '__main__':
            The testing code
            ...
```

for testing purpose. When a module is imported somewhere, '`__name__`' will be created in the module's namespace, to save the name of the module file. So when a module is imported, the testing code will be ignored by the interpreter, because a module's name, `__name__` could never be '`__main__`'. However, if we directly run the module as an executable file, like `./dial_tone.py`, or using `python dial_tone.py` to run the script, not importing, then the module's namespace is the global namespace, Python set the `__name__` of the global namespace to `__main__` automatically. In such cases, the testing code will take effects.

`fg = build_graph()` calls the function we just defined and assign the returned flow graph instance to *fg*. The graph is then made running by the next line: `fg.start()`. The definitions for the methods '`start()`', '`stop()`' are added in the class `flow_graph`:

```
        def start (self):
            '''start graph, forking thread(s), return immediately'''
            if self.scheduler:
                raise RuntimeError, "Scheduler already running"
            self._setup_connections ()

            # cast down to gr_module_sptr
            # t = [x.block () for x in self.topological_sort (self.blocks)]
            self.scheduler = scheduler (self)
            self.scheduler.start ()

        def stop (self):
            '''tells scheduler to stop and waits for it to happen'''
            if self.scheduler:
                self.scheduler.stop ()
                self.scheduler = None
```

When we start a graph, the first thing `start()` does, is to connect the graph 'physically', by calling the `self._setup_connections()` method, which we have discussed above. Then it creates an instance of the scheduler, and uses the scheduler to control the start or stop of the graph. We have talked about `scheduler` above. It plays a key role here. But its working principle might be overdetailed for us. We should feel safe to control the running of a flow graph using the methods `start()`, `stop()` in Python.

`raw_input()` is Python's build-in function, like `print`. It reads the user's input from the standard input device. For example `password = raw_input('Please input your password:')`, will print "please input your password:" on the screen and waits for the user's input. The input string is then saved in the variable 'password'. In our example, the program just waits for the user's input to stop running. Otherwise it will run forever.

7 conclusion

In this article, we place a bulk of comments onto a very simple example. We've analyzed how the flow graph mechanism brings great convenience and flexibility to the GNU Radio programming. Some of the stories hidden from the scenes are also investigated to show how the mechanism is organized.

Some of the materials covered in this article may be more than necessary for us to do programming in GNU Radio at the Python level. But it should be helpful for us to understand the whole framework.

APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio import audio
```

```
def build_graph ():
    sampling_freq = 32000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect (src0, (dst, 0))
    fg.connect (src1, (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

References

[1] **Python on-line tutorials**, http://www.python.org/doc/current/tut/

[2] Eric Blossom, **Exploring GNU Radio**,
http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

Tutorial 6: Graph, Blocks & Connecting

Dawei Shen[1]

June 7, 2005

Abstract

In tutorial 5, we have seen the family of flow graph classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a flow graph accommodates several signal sources, sinks and signal processing blocks then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

Contents

1 Overview

In tutorial 5, we have seen the family of `flow graph' classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a `flow graph' accommodates several signal sources, sinks and signal processing blocks, then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

Another example, `dial tone', is chosen for illustration purpose. This is a very simple `Hello World!' style example. But it should be enough to demonstrate the strength and beauty of the graph mechanism in GNU Radio. In this example, we simply generate two sine waves of different frequency and play the tones through the sound card. Only the signal source and sink are involved, without real signal processing.

The source code of this example is located at: '`gnuradio-examples/python/audio/dial_tone.py`'. Please refer to appendix A to make sure we have the same version of the code. To run this example, no USRP board is required, but the sound card equipped on your computer. Simply input the command `$./dial_tone.py` in your shell, you can hear clear tones from your speaker.

A few Python programming tips will also be reminded along the way.

2 Defining a function build_graph()

Two modules *gr* and *audio* are imported first. Then a function `build_graph()`, with no arguments, is defined:

```
def build_graph ():
```

build_graph () is a global function within the module dial_tone.py, slightly different with the __init__() we talked about last time, which is a method belonging to the class wfm_rx_graph.

Next two constants are defined. Just a reminder: don't forget the indentation!

```
    sampling_freq = 32000
    ampl = 0.1
```

We have met sampling_freq before when we talked about the audio decimation in last tutorial. It is the data rate of the digital signal entering the sound card. For most sound cards we use, 32kHz is a fairly good choice. The sampling rate is a parameter for the signal source gr.sig_source_f we will see later. Another parameter is the amplitude of the sine waves, ampl. We set it to 0.1v in this example.

3 Creating a graph

The graph comes!

```
    fg = gr.flow_graph ()
```

flow_graph is a class defined in the module gr.flow_graph.py.

Here is a trick. Why could we use gr.flow_graph directly to refer to this class, not gr.flow_graph.flow_graph? The answer is in the initialization file `__init__.py' of the package *gr*. Let's look at its content:

```
    from gnuradio_swig_python import *
    from basic_flow_graph import *
    from flow_graph import *
    from exceptions import *
    from hier_block import *
```

We know a directory must contain a __init__.py file to let Python treat it as a *package*. The statements in __init__.py will be executed first immediately after the package is imported somewhere. In this case, when *gr* is imported, all the definitions in flow_graph are imported to the package *gr*'s global symbol table. Here `*' represents for `everything'. So we can directly access to the classes or functions defined in flow_graph.py using gr.item rather than gr.flow_graph.item.

flow_graph is derived from its `father class' basic_flow_graph, the root of all `GRAPH' related classes. basic_flow_graph describes the connections between blocks conceptually. Let's take a look at its source file located at:

```
    /usr/local/python2.4/site-packages/gnuradio/gr/basic_flow_graph.py
```

By common sense, a graph should consist of vertices as well as the edges connecting them. These two basic elements are defined as endpoint class and edge class in basic_flow_graph.py.

Part of the definition for endpoint is

```
    class endpoint (object):
        __slots__ = ['block', 'port']
        def __init__ (self, block, port):
            self.block = block
            self.port = port
```

In Python, `__slot__' is a special variable of a class. By default, all instances of classes have a `dictionary' for attribute storage: the variable `__dict__', the data type of which is `**Dictionary**'. This mechanism wastes space for objects having very few variables. The space consumption can become acute when creating large numbers of instances. The __slots__ declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because __dict__ is not created for each instance.

So from __slot__, we can see the endpoint class has two attributes: block and port. This 2-tuple can fully specify an end point (vertice) in a graph. A block can be a source, a sink, or a signal process block. One block may have multiple ports, corresponding to different vertices in the graph.

This is part of the definition for the class edge:

```
class edge (object):
    __slots__ = ['src', 'dst']
    def __init__ (self, src_endpoint, dst_endpoint):
        self.src = src_endpoint
        self.dst = dst_endpoint
```

An edge provides a directed connection between two end points. Obviously, edge has two attributes: src and dst, indicating where the signal flow starts and ends. The data type for src and dst is just the class `endpoint' we have defined.

Based on endpoint and edge, we define the class basic_flow_graph. Part of its definition is:

```
class basic_flow_graph (object):
    '''basic_flow_graph -- describe connections between blocks'''
    __slots__ = ['edge_list']
    def __init__ (self):
        self.edge_list = []
```

basic_flow_graph has only one attribute `edge_list', which is a **List** saving all the edges in the graph. It is initialized to be empty when a flow graph instance is created. The `list' will be modified only if we call the methods defined in the class basic_flow_graph, such as `connect' or `disconnect'. We will talk about these methods later.

The class flow_graph used in our example is derived from basic_flow_graph. It is defined in

```
/usr/local/python2.4/site-packages/gnuradio/gr/flow_graph.py
```

This is part of its definition:

```
class flow_graph (basic_flow_graph):
    """add physical connection info to simple_flow_graph
    """
    __slots__ = ['blocks', 'scheduler']

    def __init__ (self):
        basic_flow_graph.__init__ (self);
        self.blocks = None
        self.scheduler = None
```

The comment reveals everything. Compared with its `father class' basic_flow_graph, flow_graph adds the physical connection to it. basic_flow_graph only defines a flow graph `abstractly', preserving the information about what kind of `end points' have been used in the graph and which ones of them are connected by the directed `edges'. You can think of it as a diagram drawn on your draft paper, nothing in the real world occurs. Of course neither can you run the graph. In `flow_graph`, we connect the blocks `physically' , i.e., all blocks' inputs are connected to their upstream buffers - real buffers in the computer memory. To understand the `physical connection' more thoroughly, we can look at the following methods defined in the class `flow_graph`:

```
    def _setup_connections (self):
        """given the basic flow graph, setup all the physical connections"""
        self.validate ()
        self.blocks = self.all_blocks ()
        self._assign_details ()
        self._assign_buffers ()
        self._connect_inputs ()

    def _assign_details (self):
        ...
    def _assign_buffers (self):
        """determine the buffer sizes to use, allocate them and attach to detail"""
        ...
    def _connect_inputs (self):
        """connect all block inputs to appropriate upstream buffers"""
        ...
```

When a graph instance calls its `start()` method to run the program, the graph will first call its `_setup_connections ()` method to setup all the physical connections. It actually calls three methods: `_assign_details()`, `_assign_buffers()` and `_connect_inputs()` sequentially. The three methods finish the real connection job on a `physical' level. The required buffer size is calculated and then allocated. Finally all the blocks' input are connected to appropriate upstream buffers. The implementation details in these methods are overdetailed for us and seldom needed when we do programming with GNU Radio. We never call these methods explicitly when we play with the graph. However, I believe knowing some of the story hidden behind could help us obtain a better understanding about the operating mechanism of GNU Radio.

`flow_graph` adds two more attributes: `` `blocks' `` and `` `scheduler' ``. `blocks` is a **List** variable, keeping the list of all blocks existing in the graph. It uses the method `all_blocks()` defined in `basic_flow_graph` to get the list. Two endpoints with the same block but different ports will be counted only once. `scheduler` is an important variable controlling the running of the flow graph. Its data type is the class `scheduler`, defined in the module *gr.scheduler*. `gr.scheduler` uses Python's built-in module `threading`, to control the `starting', `stopping' or `waiting' operations of the graph.

The most distinguishing parts that `flow_graph` adds to the `basic_flow_graph` are the methods that control the running of the flow graph, such as `start()`, `stop()`, `wait()` `run()`, etc. As indicated by their names, these methods are designed to actually drive the signal flows to move on or stop in the graph. `scheduler` plays a key role in these methods.

We can imagine that for most of the time when we want to create a graph instance or derive a graph class, we should use `flow_graph`, rather than `basic_flow_graph`.

OK! We have got so many stories from this single line: `fg = gr.flow_graph ()`. I hope you haven't forgotten, we have created a graph! It's an instance of the class `flow_graph`.

One final comment, in the introduction above, we mentioned several Python's new data types, such as `**List**' `**tuple**' and `**Dictionary**'. They are actually part of the reason why Python is so powerful, flexible and efficient. But unfortunately, we won't cover all these details in this tutorial. Please refer to the Python tutorial to gather more information about them.

4 Signal sources and sinks

4.1 The sources

Let's move on. Next we create two data sources. In last tutorial, we have seen one type of source: *usrp*. It takes the USRP board as the signal input. In this example, the source is much simpler: sine waves generated by the computer:

```
src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
```

Strictly speaking, this signal source also belongs to the category of **`blocks**'. As mentioned before, in GNU Radio, all the blocks are implemented using C++. The **SWIG** provides the interfaces between Python and C++, so that in Python you can directly use the class definitions or functions implemented using C++. We will talk about how to write blocks using C++ and how to use **SWIG** to construct the interfaces in tutorial 10. That will be the best time to understand the whole story. At this point, let's forget those annoying tricks and just use those blocks safely in Python.

The best way to learn what blocks have been implemented and bundled in GNU Radio is to look at the documentation of GNU Radio generated using Doxygen. After the installation of GNU Radio, it should be located at:

```
/usr/local/share/doc/gnuradio-core-2.5cvs/html/index.html
```

It's also available on-line here.

Press the tag `Class Hierarchy' and look for the class `gr_sig_source_f`. Once you find it, open the class reference for it, you can see the class hierarchy of `gr_sig_source_f`. The `root' class is `gr_block`. All signal processing blocks are derived from `gr_block`. `gr_sync_block` is an important class derived from `gr_block`. It implements a 1:1 block with optional history and certain simplifications are made. Sources and sinks , such as `gr_sig_source_f`, are derived from `gr_sync_block`. The only thing different about them is that sources have no inputs and sinks have no outputs. All these blocks are started with the prefix `` `gr_' `` and are implemented using C++. There is some behind-the-scenes magic when we use **SWIG** to provide the interface between C++ and Python, so that we can access `gr_sig_source_f` from Python as `gr.sig_source_f()`. The suffix `` `_f' `` here, indicates the data type of the source, which is `float' in our example. All these magics will be touched again when we talk about writing blocks in C++, so don't worry if you have any confusion at this point.

Anyway, the bottom line is, we have created two source instances returned by `gr.sig_source_f()`. They can be drawn into the `graph' we created just now!

4.2 The sinks

We have seen the same sink before in the FM receiver: the audio device. We need to play the tones generated using the sound card.

```
dst = audio.sink (sampling_freq)
```

The *audio* module is located at:

```
/usr/local/lib/python2.4/site-packages/gnuradio/audio.py
```

The module *audio* is quite similar as *usrp*. It provides a wrapper for the audio sinks and sources. Again, there are some delicate magics behind the scene, so that you can create an audio sink instance in Python directly using `audio.sink()`. `` `sink()' `` and `` `source()' `` are actually functions defined in the module *audio*, which are connected to the OSS or ALSA supports for the sound cards, assuming you have installed *gr-audio-oss* or *gr-audio-alsa* modules. These two methods will return a class instance as the audio sink or source.

The story hidden from your view may be complicated, but constructing an audio sink in Python is really simple. We have done it! Let's forget those annoying details.

5 Connecting

Finally, we connect the blocks we have defined together to complete our flow graph:

```
fg.connect (src0, (dst, 0))
fg.connect (src1, (dst, 1))
```

The usage of the `connect()' method is quite straightforward. But there are some issues worth mentioning. Let's visit the definition of the class `basic_flow_graph` again. This is the definition for the method `connect()`:

```
def connect (self, *points):
    '''connect requires two or more arguments that can be coerced to
    endpoints. If more than two arguments are provided, they are
    connected together successively.
    '''
    if len (points) < 2:
        raise ValueError, ("connect requires at least two endpoints;
        %d provided." % (len (points),))
    for i in range (1, len (points)):
        self._connect (points[i-1], points[i])
```

The `*' before the argument `points` is a special feature of Python's function. It means the function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a **tuple**. Before the variable number of arguments, zero or more normal arguments may occur. So the `connect()` method can actually accept more than 2 arguments. If more than two blocks are provided to `connect()` as arguments, they will be connected one by one successively, on which the code is quite clear.

Another issue hidden behind is, when we connect the points, the points are first `coerced' to be an `endpoint' instance. It's necessary to recast the argument first to make it consistent with the class `endpoint`. Let's see the definition of the method `coerce_endpoint()`:

```
def coerce_endpoint (x):
    if isinstance (x, endpoint):
        return x
    elif isinstance (x, types.TupleType) and len (x) == 2:
        return endpoint (x[0], x[1])
    elif hasattr (x, 'block'):          # assume it's a block
        return endpoint (x, 0)
    elif isinstance(x, hier_block.hier_block_base):
        return endpoint (x, 0)
    else:
        raise ValueError, "Not coercible to endpoint: %s" % (x,)
```

It's obvious that we don't need to provide an entire `endpoint` instance as the argument to the `connect()` method (in fact doing this will make coding messy). We can simply provide a 2-tuple as the arguments, i.e. (block name, port No.). The `coerce_endpoint()` method will convert it to an `endpoint` instance. Like in our example, we refer to the end points as `(dst, 0)` and `(dst, 1)`. If we only provide the block instance without the port number, the `coerce_endpoint`

`()` method will add 0 as the default port number and return an `endpoint` instance, like the `src` in our example. It's equivalent to say `(src, 0)`. For blocks with only one port, it's convenient to directly use the block name as the argument for `connect()`.

Note that as mentioned above, at this point, the graph is only created and connected logically, not physically. We only have a diagram on the draft paper.

Finally, the global function `build_graph()` returns the the flow graph instance `fg` we have created.

6 Run the program

Now it's time to make the flow graph running.

```
if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

What's the first line mean? Every module has an attribute `__name__', to indicate the name of the current module. Many modules contain a paragraph of code like this:

```
if __name__ == '__main__':
    The testing code
    ...
```

for testing purpose. When a module is imported somewhere, `__name__' will be created in the module's namespace, to save the name of the module file. So when a module is imported, the testing code will be ignored by the interpreter, because a module's name, `__name__` could never be `__main__'. However, if we directly run the module as an executable file, like `./dial_tone.py`, or using `python dial_tone.py` to run the script, not importing, then the module's namespace is the global namespace, Python set the `__name__` of the global namespace to `__main__` automatically. In such cases, the testing code will take effects.

`fg = build_graph()` calls the function we just defined and assign the returned flow graph instance to *fg*. The graph is then made running by the next line: `fg.start()`. The definitions for the methods `start()', `stop()' are added in the class `flow_graph`:

```
def start (self):
    '''start graph, forking thread(s), return immediately'''
    if self.scheduler:
        raise RuntimeError, "Scheduler already running"
    self._setup_connections ()

    # cast down to gr_module_sptr
    # t = [x.block () for x in self.topological_sort (self.blocks)]
    self.scheduler = scheduler (self)
    self.scheduler.start ()

def stop (self):
    '''tells scheduler to stop and waits for it to happen'''
    if self.scheduler:
        self.scheduler.stop ()
        self.scheduler = None
```

When we start a graph, the first thing `start()` does, is to connect the graph `physically', by calling the `self._setup_connections()` method, which we have discussed above. Then it creates an instance of the scheduler, and uses the scheduler to control the start or stop of the graph. We have talked about `scheduler` above. It plays a key role here. But its working principle might be overdetailed for us. We should feel safe to control the running of a flow graph using the methods `start()`, `stop()` in Python.

`raw_input()` is Python's build-in function, like `print`. It reads the user's input from the standard input device. For example `password = raw_input('Please input your password:')`, will print "please input your password:" on the screen and waits for the user's input. The input string is then saved in the variable `password'. In our example, the program just waits for the user's input to stop running. Otherwise it will run forever.

7 conclusion

In this article, we place a bulk of comments onto a very simple example. We've analyzed how the flow graph mechanism brings great convenience and flexibility to the GNU Radio programming. Some of the stories hidden from the scenes are also investigated to show how the mechanism is organized.

Some of the materials covered in this article may be more than necessary for us to do programming in GNU Radio at the Python level. But it should be helpful for us to understand the whole framework.

APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 32000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect (src0, (dst, 0))
    fg.connect (src1, (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

References

[1]

Python on-line tutorials, http://www.python.org/doc/current/tut/

[2]

Eric Blossom, **Exploring GNU Radio**,

http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

---

Footnotes:

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

---

Tutorial 7: Exploring the FM receiver

Dawei Shen[*]

July 12, 2005

Abstract

In tutorial 5, we skip the discussion on how the FM signal is demodulated, leaving the big '`guts`' as a black box. In this article, the real signal processing techniques for demodulating the FM signal are introduced. We will dig into this black box and see how the signal is processed in the software world.

1 Overview

In previous tutorials, we have introduced the hardware setup of GNU Radio and some programming tips, which form the basis of a real application in the 'soft world'. In this article, we will investigate how the broadcast FM signal is demodulated. The FM receiver is a typical example in GNU Radio. You are able to hear strong stations using just a piece of wire. In tutorial 5, we skipped the introduction to the big box '`guts`', where the real magics of the FM detection are. We will show how the signal is processed from the air to the sound card in this tutorial.

2 From the air to the computer, from real to complex

In tutorial 3 and 4, we have discussed the operations on the USRP, especially the role of the digital down converter (DDC). Basically what the USRP does is to select the part of the spectrum we are interested in and decimate the digital sequence by some factor N. The resulting signal is complex (`gr_complex`) with I/Q two channels. So after we finish writing these lines

```
src = usrp.source_c (0, decim)     # decim = 250, so data rate (quad_rate) is 256kHz
src.set_rx_freq (0, IF_freq)       # IF_freq = our input - 128MHz
src.set_pga(0,20)
```

we've got a 'complex' signal, with a data rate 256k samples per second. We name it 'quadrature rate' - `quad_rate` because the complex signal has I/Q quadrature components.

The IF frequency we choose is an interesting and useful point here. `IF_freq` is equal to the user's input minus 128MHz, as the line in function `parseargs()` indicates:

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

```
return freq1 - 128e6
```

`wfm_rcv_gui` was written assuming that there was no RF front end doing any down conversion. The A/D sample rate is 64M. The Nyquist zones are therefore:

```
[0, 32M]    normal
[32M, 64M]  inverted
[64M, 96M]  normal
[96M, 128M] inverted
```

etc.

The problem is that 96M, one of the folding points, occurs right in the middle of the broadcast band. This means that 95.0 and 97.0 both alias down to the same frequency and can't be distinguished from each other. `freq1 -` 128M will give a valid frequency if freq1 is >= 96M. Bottom line: it's a kludge that sort of works.

3 Getting the instantaneous frequency, from complex to real

It's time to dig into the heart of the '`guts`' now, which we left as a big black box in tutorial 5.

```
guts = blks.wfm_rcv (self, quad_rate, audio_decimation)
```

`blks` is a package in `gnuradio`. It almost does nothing but refers to another package `blksimpl`. `wfm_rcv` is a class defined in `/gnuradio/blksimpl/wfm_rcv.py`, which is real 'processor' of the FM receiver. The source code is appended at the end.

`wfm_rcv` is derived from `gr.hier_block`. `gr.hier_block` describes a series of blocks in tandem in a flow graph. It assumes that there is at most a single block at the head of the chain and a single block at the end of the chain. Either head or tail may be None indicating a sink or source respectively. `hier_block` could be recognized a sub-graph consisting of several blocks connected one after another. To construct a `hier_block`, we need to specify the flow graph that contains this hierarchical block, the first and the last block in the signal processing chain. A hierarchical block could be treated as a common block, which could be placed and connected in a flow graph, like these lines demonstrate:

```
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

The 'head' block in the chain is `fm_demod`, the instance of `gr.quadrature_demod_cf`. To understand the real work within it, we should know a bit about how FM signals are generated. With FM, the instantaneous frequency of the transmitted waveform is varied as a function of the input signal. The instantaneous frequency at any time is given by the following formula:

```
f(t) = k * m(t) + fc
```

m(t) is the input signal, k is a constant that controls the frequency sensitivity and fc is the frequency of the carrier (for example, 100.1MHz). So to recover m(t), two steps are needed. First we need to remove the carrier fc, then we're left with a baseband signal that has an instantaneous frequency proportional to the original message m(t). The second step is obviously to compute the instantaneous frequency of the baseband signal. Thus, our challenge is to find a way to remove the carrier and compute the instantaneous frequency. Removing the carrier has been done on the FPGA, via the digital down converter (DDC), as introduced in tutorial 3 and 4. We have explained why we tune to (fc - 128MHz) in the preceding section. The resulting signal coming into the 'guts' has already become a baseband signal and the remaining task is to calculate its instantaneous frequency. If we integrate frequency, we get phase, or angle. Conversely, differentiating phase with respect to time gives frequency. These are the key insights we use to build the receiver.

We use the `gr.quadrature_demod_cf` block for computing the instantaneous frequency of the baseband signal. We approximate differentiating the phase by determining the angle between adjacent samples. Recall that the digital down converter produces complex numbers on its output. Using a bit more trigonometry, we can determine the angle between two subsequent samples by multiplying one by the complex conjugate of the other and then taking the arc tangent of the product. Once you know what you want, it doesn't take much code. `gr_quadrature_demod_cf.cc` contains the C++ implementation of this block. We will talk about how to write a signal processing block using C++ in detail in tutorial 10 and 11. But it's useful to give a shot at the code now. The bulk of the signal processing is the three-line loop in `sync_work()` function.

```
Part of gr_quadrature_demod_cf.cc
...

int
gr_quadrature_demod_cf::sync_work (
    int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  gr_complex *in = (gr_complex *) input_items[0];
  float *out = (float *) output_items[0];
  in++;          // ensure that in[-1] is valid
  for (int i = 0; i < noutput_items; i++){
    gr_complex product = in[i] * conj (in[i-1]);
    out[i] = d_gain * arg (product);
  }
  return noutput_items;
}
```

A helpful diagram for the FM receiver can be found here.

```
fm_demod_gain = quad_rate/(2*math.pi*max_dev)
```

Note that `fm_demod_gain` can be treated as a constant controlling the volume. Its real value doesn't matter. Here `arg (product)` is the phase difference between adjacent samples, if we divide it by the sample interval, i.e. multiply the data rate `quad_rate`, we get the radian frequency ω, which gives us the instantaneous frequency f if we further divide ω by 2π. Finally we normalize the frequency by `max_dev`.

4 Deemphasizer

The second block in the chain is a deemphasizer `deemph`, an instance of the class `fm_deemph`. `fm_deemph` is defined in `fm_emph.py`, also located in the package `blksimpl`.

What is deemphasis? Let's introduce it briefly. It has been theoretically proved that, in FM detector, the power of the output noise increases with the frequency quadratically. However, for most practical signals, such as human voice and music, the power of the signal decreases significantly as frequency increases. As a result, the signal noise ratio (SNR) at the high frequency end usually becomes unbearable. To circumvent this effect, people introduce 'preemphasis' and 'deemphasis' into the FM system. At the transmitter, we use proper preemphasis circuits to manually amplify the high frequency components, and do the converse operations at the receiver to recover the original power distribution of the signal. As a result, we improve the SNR effectively.

In the analog world, a simple first order RLC circuit usually suffices for preemphasis and deemphasis. Here is a nice plot of their transfer functions. In our digital signal processing, a first order IIR filter could be the right choice.

```
Part of fm_emph.py
...

#           1
# H(s) = -------
#         1 + s
#
# tau is the RC time constant.
# critical frequency: w_p = 1/tau
#
# We prewarp and use the bilinear z-transform to get our IIR coefficients.
# See "Digital Signal Processing: A Practical Approach" by Ifeachor and Jervis
#
class fm_deemph(gr.hier_block):
    """
    FM Deemphasis IIR filter.
    """
    def __init__(self, fg, fs, tau=75e-6):
        """
        @type fs: float
        @param tau: Time constant in seconds (75us in US, 50us in EUR)
        @type tau: float
        """
        w_p = 1/tau
        w_pp = math.tan (w_p / (fs * 2)) # prewarped analog freq

        a1 = (w_pp - 1)/(w_pp + 1)
        b0 = w_pp/(1 + w_pp)
        b1 = b0

        btaps = [b0, b1]
        ataps = [1, a1]

        deemph = gr.iir_filter_ffd(btaps, ataps)
        gr.hier_block.__init__(self, fg, deemph, deemph)
```

We start from the transfer function of the analog filter as prototype, and use bilinear transformation to get the digital IIR filter. Note that `gr.iir_filter_ffd` is the IIR filter block with float input, float output and double taps.

5 Audio FIR decimation filter

After passing the deemphasizer, how does the signal look like now? First, it's a real signal with a data rate of 256kHz. Second, it's a baseband signal, with effective frequency range from 0 to about 100kHz, containing all the frequency components of a FM station.

As a side note, the bandwidth of a FM station is usually around 2 * 100kHz. This also explains why we choose 256kHz as the quadrature rate (the decimation rate on the USRP is chosen to be 250). A sample rate of 256kHz is just suitable for the 200kHz bandwidth, without losing any spectrum information. Maybe you have noticed, in the FM receiver, we never use any low-pass filtering operation to 'pick out' the FM station we are interested in. Actually this is done implicitly in the digital down converter (DDC) on the USRP. Recall that digital down converter can be regarded as a low-pass FIR filter followed by a downsampler. As a result, the target station is picked out then spread out to the entire digital spectrum after decimation. Because we choose the right decimation rate, we have eventually done a lot! The bottom line is, we are operating on a single FM station after the signal goes through the USRP.

Now we need to resolve two issues. First, the signal rate is 256kHz now, much higher than what the sound card can adopt. The PC sound cards usually sample up to 96,000 Hz maximum. Second, the 100kHz spectrum contains several channels, L + R, L - R, pilot tones, etc. To keep our life simpler, we just want to design a mono receiver low-passing only the L + R signal. So clearly, an FIR decimation filter is exactly what we want.

To make things clearer, here is a brief introduction to the FM signal band. From 0 to about 16kHz is the left plus right (L + R) audio. The peak at 19kHz is the stereo pilot tone. The left minus right (L - R) stereo information is centered at 2x the pilot (38kHz) and is AM-modulated on top of the FM. Additional subcarriers are sometimes found in the region of 57kHz - 96kHz. We can use the GNU Radio built-in fft block with GUI supports to view the spectrum of the demodulated signal (details will be introduced in tutorial 8). Here is a nice real plot given by Eric. Here is a good illustration of the FM band.

OK, now let's design the FIR decimation filter. The GNU Radio block `gr.fir_filter_fff` gives us an FIR filter with float input, float output, and float taps. Its constructor takes two arguments: the first one is the decimation factor, the second one is the filter coefficients (taps) vector.

```
    self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)
```

If we need an FIR filter without changing the data rate, then we just simply set the decimation rate to be 1. If we need an interpolation filter rather than a decimation filter, then the GNU Radio block `gr.interp_fir_filter_xxx` is what we should choose.

The filter coefficients `audio_coeffs` are obtained using the FIR filter design block `gr.firdes`. `low_pass()` is a static public function defined in the class `gr_firdes`. Similarly, it also has `high_pass()`, `band_pass()`, `band_reject()` functions. We use these functions to design the FIR filter taps, provided the filter parameters and specifications. For example, the syntax for design a low-pass filter is:

```
vector< float > gr_firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76
                                     )   [static]
```

The meaning of each argument is quite obvious. Note that `beta` is a parameter for Kaiser window. In our example, we select the audio decimation factor (`audio_decimation`) to be 8, so that the resulting data rate for the sound card is 32kHz. We are only interested in the L + R audio from 0 to 16kHz, so we low pass the output of the quadrature demodulator with a cutoff frequency of 16kHz. This gives us a monaural output that we connect to the sound card outputs. In our example, we choose the cutoff frequency as 15kHz and transition band as 1kHz, which is reasonable.

```
width_of_transition_band = audio_rate / 32
audio_coeffs = gr.firdes.low_pass (volume,         # gain                      (20)
                                   quad_rate,      # sampling rate          (32kHz)
                                   audio_rate/2 - width_of_transition_band, (15kHz)
                                   width_of_transition_band,                (16kHz)
                                   gr.firdes.WIN_HAMMING)
```

OK! Our FM receiver is complete! The signal is at the door of the sound card and is ready to be played. Note that the usage of FIR filters, as well as multirate processing is very important in the digital signal processing.

Finally, we connect these blocks and call the `__init__()` method of `gr.hier_block` to complete the `__init__()` method of the `wfm_rcv` class. Here we need to specify the head and the tail of the pipeline.

```
fg.connect (self.fm_demod, self.deemph, self.audio_filter)
gr.hier_block.__init__(self,
                       fg,
                       self.fm_demod,       # head of the pipeline
                       self.audio_filter)   # tail of the pipeline
```

6 Conclusion

In this article, we have introduced the FM detection techniques and how they are implemented using GNU Radio. Now we can see GNU Radio is really a nice system, providing us so many powerful tools and flexible ways to construct a real application. In next tutorial, we will wrap up the explanation of `wfm_rcv_gui.py`, with an emphasis on the GNU Radio GUI tools.

APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio.blksimpl.fm_emph import fm_deemph
import math

class wfm_rcv(gr.hier_block):
    def __init__ (self, fg, quad_rate, audio_decimation):
        """
        Hierarchical block for demodulating a broadcast FM signal.

        The input is the downconverted complex baseband signal (gr_complex).
        The output is the demodulated audio (float).

        @param fg: flow graph.
        @type fg: flow graph
        @param quad_rate: input sample rate of complex baseband input.
        @type quad_rate: float
        @param audio_decimation: how much to decimate quad_rate to get to audio.
        @type audio_decimation: integer
        """
        volume = 20.

        max_dev = 75e3
        fm_demod_gain = quad_rate/(2*math.pi*max_dev)
        audio_rate = quad_rate / audio_decimation
        # We assign to self so that outsiders can grab the demodulator
        # if they need to.  E.g., to plot its output.
        #
        # input: complex; output: float
        self.fm_demod = gr.quadrature_demod_cf (fm_demod_gain)

        # input: float; output: float
        self.deemph = fm_deemph (fg, quad_rate)

        # compute FIR filter taps for audio filter
        width_of_transition_band = audio_rate / 32
        audio_coeffs = gr.firdes.low_pass (volume,         # gain
                                           quad_rate,      # sampling rate
                                           audio_rate/2 - width_of_transition_band,
                                           width_of_transition_band,
                                           gr.firdes.WIN_HAMMING)
        # input: float; output: float
        self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)

        fg.connect (self.fm_demod, self.deemph, self.audio_filter)

        gr.hier_block.__init__(self,
                               fg,
                               self.fm_demod,       # head of the pipeline
                               self.audio_filter)   # tail of the pipeline
```

References

[1] Eric Blossom, **Listening to FM Radio in Software, Step by Step**, http://www.linuxjournal.com/article/7505

Tutorial 7: Exploring the FM receiver

Dawei Shen[1]

July 12, 2005

Abstract

In tutorial 5, we skip the discussion on how the FM signal is demodulated, leaving the big `guts' as a black box. In this article, the real signal processing techniques for demodulating the FM signal are introduced. We will dig into this black box and see how the signal is processed in the software world.

Contents

1 Overview

In previous tutorials, we have introduced the hardware setup of GNU Radio and some programming tips, which form the basis of a real application in the `soft world'. In this article, we will investigate how the broadcast FM signal is demodulated. The FM receiver is a typical example in GNU Radio. You are able to hear strong stations using just a piece of wire. In tutorial 5, we skipped the introduction to the big box `guts', where the real magics of the FM detection are. We will show how the signal is processed from the air to the sound card in this tutorial.

2 From the air to the computer, from real to complex

In tutorial 3 and 4, we have discussed the operations on the USRP, especially the role of the digital down converter (DDC). Basically what the USRP does is to select the part of the spectrum we are interested in and decimate the digital sequence by some factor N. The resulting signal is complex (`gr_complex`) with I/Q two channels. So after we finish writing these lines

```
    src = usrp.
source_c (0, decim)    # decim = 250, so data rate (quad_rate) is 256kHz
    src.set_rx_freq (0, IF_freq)      # IF_freq = our input - 128MHz
    src.set_pga(0,20)
```

we've got a `complex' signal, with a data rate 256k samples per second. We name it `quadrature rate' - `quad_rate` because the complex signal has I/Q quadrature components.

The IF frequency we choose is an interesting and useful point here. `IF_freq` is equal to the user's input minus 128MHz, as the line in function `parseargs()` indicates:

```
    return freq1 - 128e6
```

wfm_rcv_gui was written assuming that there was no RF front end doing any down conversion. The A/D sample rate is 64M. The Nyquist zones are therefore:

```
[0, 32M]    normal
[32M, 64M]  inverted
[64M, 96M]  normal
[96M, 128M] inverted
```

etc.

The problem is that 96M, one of the folding points, occurs right in the middle of the broadcast band. This means that 95.0 and 97.0 both alias down to the same frequency and can't be distinguished from each other. freq1 - 128M will give a valid frequency if freq1 is > = 96M. Bottom line: it's a kludge that sort of works.

3 Getting the instantaneous frequency, from complex to real

It's time to dig into the heart of the `guts' now, which we left as a big black box in tutorial 5.

```
guts = blks.wfm_rcv (self, quad_rate, audio_decimation)
```

blks is a package in gnuradio. It almost does nothing but refers to another package blksimpl. wfm_rcv is a class defined in /gnuradio/blksimpl/wfm_rcv.py, which is real `processor' of the FM receiver. The source code is appended at the end.

wfm_rcv is derived from gr.hier_block. gr.hier_block describes a series of blocks in tandem in a flow graph. It assumes that there is at most a single block at the head of the chain and a single block at the end of the chain. Either head or tail may be None indicating a sink or source respectively. hier_block could be recognized a sub-graph consisting of several blocks connected one after another. To construct a hier_block, we need to specify the flow graph that contains this hierarchical block, the first and the last block in the signal processing chain. A hierarchical block could be treated as a common block, which could be placed and connected in a flow graph, like these lines demonstrate:

```
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

The `head' block in the chain is fm_demod, the instance of gr.quadrature_demod_cf. To understand the real work within it, we should know a bit about how FM signals are generated. With FM, the instantaneous frequency of the transmitted waveform is varied as a function of the input signal. The instantaneous frequency at any time is given by the following formula:

```
f(t) = k * m(t) + fc
```

m(t) is the input signal, k is a constant that controls the frequency sensitivity and fc is the frequency of the carrier (for example, 100.1MHz). So to recover m(t), two steps are needed. First we need to remove the carrier fc, then we're left with a baseband signal that has an instantaneous frequency proportional to the original message m(t). The second step is obviously to compute the instantaneous frequency of the baseband signal. Thus, our challenge is to find a way to remove the carrier and compute the instantaneous frequency. Removing the carrier has been done on the FPGA, via the digital down converter (DDC), as introduced in tutorial 3 and 4. We have explained why we tune to (fc - 128MHz) in the preceding section. The resulting signal coming into the `guts' has already become a baseband signal and the remaining task is to calculate its instantaneous frequency. If we integrate frequency, we get phase, or angle. Conversely, differentiating phase with respect to time gives frequency. These are the key insights we use to build the receiver.

We use the gr.quadrature_demod_cf block for computing the instantaneous frequency of the baseband signal. We approximate differentiating the phase by determining the angle between adjacent samples. Recall that the digital down converter produces complex numbers on its output. Using a bit more trigonometry, we can determine the angle between two subsequent samples by multiplying one by the complex conjugate of the other and then taking the arc tangent of the product. Once you know what you want, it doesn't take much code. `gr_quadrature_demod_cf.cc` contains the C++ implementation of this block. We will talk about how to write a signal processing block using C++ in detail in tutorial 10 and 11. But it's useful to give a shot at the code now. The bulk of the signal processing is the three-line loop in `sync_work()` function.

```
Part of gr_quadrature_demod_cf.cc
...

int
gr_quadrature_demod_cf::sync_work (
    int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  gr_complex *in = (gr_complex *) input_items[0];
  float *out = (float *) output_items[0];
  in++;         // ensure that in[-1] is valid
  for (int i = 0; i < noutput_items; i++){
    gr_complex product = in[i] * conj (in[i-1]);
    out[i] = d_gain * arg (product);
  }
  return noutput_items;
}
```

A helpful diagram for the FM receiver can be found here.

```
     fm_demod_gain = quad_rate/(2*math.pi*max_dev)
```

Note that `fm_demod_gain` can be treated as a constant controlling the volume. Its real value doesn't matter. Here `arg (product)` is the phase difference between adjacent samples, if we divide it by the sample interval, i.e. multiply the data rate `quad_rate`, we get the radian frequency ω, which gives us the instantaneous frequency f if we further divide ω by 2π. Finally we normalize the frequency by `max_dev`.

4 Deemphasizer

The second block in the chain is a deemphasizer `deemph`, an instance of the class `fm_deemph`. `fm_deemph` is defined in `fm_emph.py`, also located in the package `blksimpl`.

What is deemphasis? Let's introduce it briefly. It has been theoretically proved that, in FM detector, the power of the output noise increases with the frequency quadratically. However, for most practical signals, such as human voice and music, the power of the signal decreases significantly as frequency increases. As a result, the signal noise ratio (SNR) at the high frequency end usually becomes unbearable. To circumvent this effect, people introduce `preemphasis' and `deemphasis' into the FM system. At the transmitter, we use proper preemphasis circuits to manually amplify the high frequency components, and do the converse operations at the receiver to recover the original power distribution of the signal. As a result, we improve the SNR effectively.

In the analog world, a simple first order RLC circuit usually suffices for preemphasis and deemphasis. Here is a nice plot of their transfer functions. In our digital signal processing, a first order IIR filter could be the right choice.

```
Part of fm_emph.py
```

```
...

#           1
# H(s) = -------
#         1 + s
#
# tau is the RC time constant.
# critical frequency: w_p = 1/tau
#
# We prewarp and use the bilinear z-transform to get our IIR coefficients.
# See "Digital Signal Processing: A Practical Approach" by Ifeachor and Jervis
#
class fm_deemph(gr.hier_block):
    """
    FM Deemphasis IIR filter.
    """
    def __init__(self, fg, fs, tau=75e-6):
        """
        @type fs: float
        @param tau: Time constant in seconds (75us in US, 50us in EUR)
        @type tau: float
        """
        w_p = 1/tau
        w_pp = math.tan (w_p / (fs * 2)) # prewarped analog freq

        a1 = (w_pp - 1)/(w_pp + 1)
        b0 = w_pp/(1 + w_pp)
        b1 = b0

        btaps = [b0, b1]
        ataps = [1, a1]

        deemph = gr.iir_filter_ffd(btaps, ataps)
        gr.hier_block.__init__(self, fg, deemph, deemph)
```

We start from the transfer function of the analog filter as prototype, and use bilinear transformation to get the digital IIR filter. Note that `gr.iir_filter_ffd` is the IIR filter block with float input, float output and double taps.

5 Audio FIR decimation filter

After passing the deemphasizer, how does the signal look like now? First, it's a real signal with a data rate of 256kHz. Second, it's a baseband signal, with effective frequency range from 0 to about 100kHz, containing all the frequency components of a FM station.

As a side note, the bandwidth of a FM station is usually around 2 * 100kHz. This also explains why we choose 256kHz as the quadrature rate (the decimation rate on the USRP is chosen to be 250). A sample rate of 256kHz is just suitable for the 200kHz bandwidth, without losing any spectrum information. Maybe you have noticed, in the FM receiver, we never use any low-pass filtering operation to `pick out' the FM station we are interested in. Actually this is done implicitly in the digital down converter (DDC) on the USRP. Recall that digital down converter can be regarded as a low-pass FIR filter followed by a downsampler. As a result, the target station is picked out then spread out to the entire digital spectrum after decimation. Because we choose the right decimation rate, we have eventually done a lot! The bottom line is, we are operating on a single FM station after the signal goes through the USRP.

Now we need to resolve two issues. First, the signal rate is 256kHz now, much higher than what the sound card can adopt.

The PC sound cards usually sample up to 96,000 Hz maximum. Second, the 100kHz spectrum contains several channels, L + R, L - R, pilot tones, etc. To keep our life simpler, we just want to design a mono receiver low-passing only the L + R signal. So clearly, an FIR decimation filter is exactly what we want.

To make things clearer, here is a brief introduction to the FM signal band. From 0 to about 16kHz is the left plus right (L + R) audio. The peak at 19kHz is the stereo pilot tone. The left minus right (L - R) stereo information is centered at 2x the pilot (38kHz) and is AM-modulated on top of the FM. Additional subcarriers are sometimes found in the region of 57kHz - 96kHz. We can use the GNU Radio built-in fft block with GUI supports to view the spectrum of the demodulated signal (details will be introduced in tutorial 8). Here is a nice real plot given by Eric. Here is a good illustration of the FM band.

OK, now let's design the FIR decimation filter. The GNU Radio block `gr.fir_filter_fff` gives us an FIR filter with float input, float output, and float taps. Its constructor takes two arguments: the first one is the decimation factor, the second one is the filter coefficients (taps) vector.

```
    self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)
```

If we need an FIR filter without changing the data rate, then we just simply set the decimation rate to be 1. If we need an interpolation filter rather than a decimation filter, then the GNU Radio block `gr.interp_fir_filter_xxx` is what we should choose.

The filter coefficients `audio_coeffs` are obtained using the FIR filter design block `gr.firdes`. `low_pass()` is a static public function defined in the class `gr_firdes`. Similarly, it also has `high_pass()`, `band_pass()`, `band_reject()` functions. We use these functions to design the FIR filter taps, provided the filter parameters and specifications. For example, the syntax for design a low-pass filter is:

```
vector< float > gr_firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76
                                     )   [static]
```

The meaning of each argument is quite obvious. Note that `beta` is a parameter for Kaiser window. In our example, we select the audio decimation factor (`audio_decimation`) to be 8, so that the resulting data rate for the sound card is 32kHz. We are only interested in the L + R audio from 0 to 16kHz, so we low pass the output of the quadrature demodulator with a cutoff frequency of 16kHz. This gives us a monaural output that we connect to the sound card outputs. In our example, we choose the cutoff frequency as 15kHz and transition band as 1kHz, which is reasonable.

```
width_of_transition_band = audio_rate / 32
audio_coeffs = gr.firdes.low_pass (volume,          # gain                    (20)
                                   quad_rate,       # sampling rate           (32kHz)
                                   audio_rate/2 - width_of_transition_band,  (15kHz)
                                   width_of_transition_band,                  (16kHz)
                                   gr.firdes.WIN_HAMMING)
```

OK! Our FM receiver is complete! The signal is at the door of the sound card and is ready to be played. Note that the usage of FIR filters, as well as multirate processing is very important in the digital signal processing.

Finally, we connect these blocks and call the `__init__()` method of `gr.hier_block` to complete the `__init__()` method of the `wfm_rcv` class. Here we need to specify the head and the tail of the pipeline.

```
fg.connect (self.fm_demod, self.deemph, self.audio_filter)
gr.hier_block.__init__(self,
                       fg,
                       self.fm_demod,       # head of the pipeline
                       self.audio_filter)   # tail of the pipeline
```

6 Conclusion

In this article, we have introduced the FM detection techniques and how they are implemented using GNU Radio. Now we can see GNU Radio is really a nice system, providing us so many powerful tools and flexible ways to construct a real application. In next tutorial, we will wrap up the explanation of `wfm_rcv_gui.py`, with an emphasis on the GNU Radio GUI tools.

APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio.blksimpl.fm_emph import fm_deemph
import math

class wfm_rcv(gr.hier_block):
    def __init__ (self, fg, quad_rate, audio_decimation):
        """
        Hierarchical block for demodulating a broadcast FM signal.

        The input is the downconverted complex baseband signal (gr_complex).
        The output is the demodulated audio (float).

        @param fg: flow graph.
        @type fg: flow graph
        @param quad_rate: input sample rate of complex baseband input.
        @type quad_rate: float
        @param audio_decimation: how much to decimate quad_rate to get to audio.
        @type audio_decimation: integer
        """
        volume = 20.

        max_dev = 75e3
        fm_demod_gain = quad_rate/(2*math.pi*max_dev)
        audio_rate = quad_rate / audio_decimation


        # We assign to self so that outsiders can grab the demodulator
        # if they need to.  E.g., to plot its output.
        #
        # input: complex; output: float
        self.fm_demod = gr.quadrature_demod_cf (fm_demod_gain)

        # input: float; output: float
        self.deemph = fm_deemph (fg, quad_rate)

        # compute FIR filter taps for audio filter
        width_of_transition_band = audio_rate / 32
        audio_coeffs = gr.firdes.low_pass (volume,         # gain
                                           quad_rate,      # sampling rate
                                           audio_rate/2 - width_of_transition_band,
                                           width_of_transition_band,
                                                  gr.firdes.WIN_HAMMING)
        # input: float; output: float
        self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)

        fg.connect (self.fm_demod, self.deemph, self.audio_filter)

        gr.hier_block.__init__(self,
                               fg,
                               self.fm_demod,       # head of the pipeline
                               self.audio_filter)   # tail of the pipeline
```

References

[1]
Eric Blossom, **Listening to FM Radio in Software, Step by Step**, http://www.linuxjournal.com/article/7505

Footnotes:

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

Tutorial 8: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line – Part II

Dawei Shen*

July 13, 2005

Abstract

Let's continue our discussion on the FM example `wfm_rcv_gui.py`. The usage of the GUI tools in GNU Radio, which are built upon wxPython, will be shown. We will also introduce some useful programming tips on argument parsing. If you are interested in using or even developing the GUI tools, this article would be helpful.

1 Overview

In this article, we will complete our discussion on the FM receiver code `wfm_rcv_gui.py`. One exciting feature of GNU Radio, is it incorporates powerful GUI tools for displaying and analyzing the signal, emulating the real spectrum analyzer and the oscillograph. We will introduce the usage of the GUI tools, which are built upon wxPython. Next we will talk about how to handle the command line arguments in Python.

2 GUI tools in GNU Radio

The most intuitive and straightforward way to analyze a signal is to display it graphically, both in time domain and frequency domain. For the applications in the real world, we have the spectrum analyzer and the oscillograph to facilitate us. Fortunately, in the software radio world, we also have such nice tools, thanks to wxPython, which provides a flexible way to construct GUI tools.

2.1 The 'Spectrum Analyzer' - fft_sink

Let's continue to read the code:

```
if 1:
    pre_demod, fft_win1 = \
               fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
```

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

```
                                              512, quad_rate)
        self.connect (src, pre_demod)
        vbox.Add (fft_win1, 1, wx.EXPAND)
```

This is the 'soft spectrum analyzer', based on fast Fourier transformation (FFT) of the digital sequence. This 'soft spectrum analyzer' is used as the signal sink. That's why it is named as 'fftsink'. It's defined in the module wxgui.fftsink.py. The function make_fft_sink_c() serves as the public interface to create an instance of the fft sink:

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

First, we should point out that in Python, a function could return multiple values. make_fft_sink_c() returns two values: block is an instance of the class fft_sink_c, defined in the same module wxgui.fftsink.py. Another special feature of Python needs to be emphasized: Python supports multiple inheritance. fft_sink_c is derived from two classes: gr.hier_block and fft_sink_base. Being a 'son' class of gr.hier_block implies that fft_sink_c can be treated as a normal block, which can be placed and connected in a flow graph, as the next line shows:

```
    self.connect (src, pre_demod)
```

block.win is obviously an attribute of block. In the definition of the class fft_sink_c, we can find its data type is the class fft_window, a subclass of wx.Window, also defined in the module wxgui.fftsink.py. We can think of it as a window that is going to be hang up on your screen. This window block.win will be used as the argument of the method vbox.Add.

2.2 How wxPython plays its role

To understand the other parts thoroughly, we need to know a little bit about wxPython, a GUI toolkit for Python. Interested readers may visit wxPython's website or tutorials' page for more information. It might be time consuming to explore all the technical details about wxPython. The good news is in GNU Radio, we can use the spectrum analyzer and oscillograph pretty much as it is. Just copy those lines anywhere you want with only a few changes.

Let's invest some time in wxPython's organism. The first thing to do is certainly to import all wxPython's components into current workspace, like the line 'import wx' does. Every wxPython application needs to derive a class from wx.App and provide an OnInit() method for it. The system calls this method as part of its startup/initialization sequence, in wx.App.__init()__. The primary purpose of OnInit() is to create the frames, windows, etc. necessary for the program to begin operation. After defining such a class, we need to instantiate an object of this class and start the application by calling its MainLoop() method, whose role is to handle the events. In our FM receiver example, where is such a class defined? Let's look at the last three lines:

```
if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

In fact, such a class, called stdapp has been created when we import the stdgui module.

```
    from gnuradio.wxgui import stdgui, fftsink
```

The final two lines again will probably be the same for all your wxPython applications. We simply create an instance of our application class, and then call its `MainLoop()` method. `MainLoop()` is the heart of the application and is where events are processed and dispatched to the various windows in the application. There are some tricks behind the scene. Don't worry about that.

Let's look at the definition of `stdapp` in `/gnuradio/wxgui/stugui.py`:

```
class stdapp (wx.App):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        self.flow_graph_maker = flow_graph_maker
        self.title = title
        # All our initialization must come before calling wx.App.__init__.
        # OnInit is called from somewhere in the guts of __init__.
        wx.App.__init__ (self)

    def OnInit (self):
        frame = stdframe (self.flow_graph_maker, self.title)
        frame.Show (True)
        self.SetTopWindow (frame)
        return True
```

`stdapp` is right the class derived from `wx.App`. Its initialization method `__init__()` takes two arguments: `flow_graph_maker`, a class belonging to the flow graph family (remember the biggest class `wfm_rx_graph` we created is derived from `gr.flow_graph`?); `title`, the title of the whole application (`WFM RX` in our example). In `OnInit()` method, these two arguments are further used to create the object of `stdframe`.

```
class stdframe (wx.Frame):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        # print "stdframe.__init__"
        wx.Frame.__init__(self, None, -1, title)

        self.CreateStatusBar (2)
        mainmenu = wx.MenuBar ()

        menu = wx.Menu ()
        item = menu.Append (200, 'E&xit', 'Exit')
        self.Bind (wx.EVT_MENU, self.OnCloseWindow, item)
        mainmenu.Append (menu, "&File")
        self.SetMenuBar (mainmenu)

        self.Bind (wx.EVT_CLOSE, self.OnCloseWindow)
        self.panel = stdpanel (self, self, flow_graph_maker)
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(vbox)
        self.SetAutoLayout(True)
        vbox.Fit(self)

    def OnCloseWindow (self, event):
        self.flow_graph().stop()
        self.Destroy ()

    def flow_graph (self):
        return self.panel.fg
```

It's worth taking a while to understand the layout of a wxPython GUI. In wxPython, a `wx.Window` is anything which can take up visual space on the computer screen. Thus, the `wx.Window` class is the base class from which all visual elements are derived – including input fields, pull-down menus, buttons, etc. The `wx.Window` class defines all the behavior common to all visual GUI elements, including positioning, sizing, showing, giving focus, etc. If we're looking for an object to represent a window on the computer screen, don't look at `wx.Window`, look at `wx.Frame` instead. `wx.Frame` is derived from `wx.Window`. It implements all behavior specific to windows on the computer's screen. A 'Frame' is a window whose size and position can (usually) be changed by the user. It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A 'Frame' can contain any window that is not a frame or dialog. So to create a 'window' on the computer screen, you create a `wx.Frame` (or one of its sub-classes, such as `wx.Dialog`), rather than a `wx.Window`.

Within a frame, you'll use a number of `wx.Window` sub-classes to flesh out the frame's contents, such as `wx.MenuBar`, `wx.StatusBar`, `wx.ToolBar`, sub-classes of `wx.Control` (eg. `wx.Button`, `wx.StaticText`, `wx.TextCtrl`, `wx.ComboBox`, etc.), or `wx.Panel`, which is a container to hold your various `wx.Control` objects. All visual elements (`wx.Window` objects and their subclasses) can hold sub-elements. A `wx.Frame` might hold a number of `wx.Panel` objects, which in turn hold a number of `wx.Button`, `wx.StaticText` and `wx.TextCtrl` objects.

In our example, `stdframe`, the subclass of `wx.Frame`, is used to create the 'frame'. We make this frame appear by 'showing' it using `frame.Show (True)`. The `SetTopWindow()` method simply tells that this frame is one of the main frames (in this case the only one) for the application. Notice the shape of the constructor of `wx.Frame`:

```
wx.Frame(Parent, Id, "title")
```

Most of the constructors in wxPython have this shape: A parent object as a first parameter and an Id in a second parameter. As shown in the example, it's possible to use respectively None and -1 as default parameters, meaning the object has no parent and respectively a system-defined Id.

In `stdframe.__init__()`, we create a `panel` and place inside the frame.

```
self.panel = stdpanel (self, self, flow_graph_maker)
```

The class `stdpanel` is derived from `wx.Panel`:

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        wx.Panel.__init__ (self, parent, -1)
        self.frame = frame

        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)

        self.fg.start ()
```

Note that `panel`'s parent is the `frame` object we create just now, meaning this panel is a subcomponent of the frame. The `frame` places the `panel` inside itself using a `wx.BoxSizer`, `vbox`. The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or nested hierarchies of either. A `wx.BoxSizer` will lay out its items in a simple row or column, depending on the orientation parameter passed to the constructor. In our example, `vbox = wx.BoxSizer(wx.VERTICAL)` tells the constructor all the items will be placed vertically. The `SetSizer()` call tells the frame which sizer to use. The call to `SetAutoLayout()` tells the frame to use the sizer to position and size your components. And finally, the call to `sizer.Fit()` tells the sizer to calculate the initial size and position for all its elements. If you are using sizers, this is the normal process you would go through to set up your window or frame's contents before it is displayed for the first time. The most important and useful method for a sizer is `add()`, it appends an item to the sizer. Its syntax looks like:

```
Add(self, item, proportion=0, flag=0)
```

'`item`' is what you wish to append to the sizer, usually a `wx.Window` object. It can also be a child sizer. `proportion` is related to whether a child of a sizer can change its size in the main orientation of the wx.BoxSizer. There are several flags defined in wx, and they are used to determine how the sizer items behave and the border of the window. `wx.EXPAND` used in our example means the item will be expanded to fill the space allotted to the item. Refer to this page for a complete description.

Have you ever had this confusion: we define a 'big' class `wfm_rcv_graph`, but where do we use it? why do we never create an instance of this class? The secret is revealed in `stdpanel.__init__()`. The instance of `wfm_rcv_graph` is created here and the flow graph is started.

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        ......
        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)
        self.fg.start ()
```

We put a panel in the frame, but what do we put in the panel? We first create a new sizer `vbox` for the panel. Note that this `vbox` is different from the one defined in `stdframe`. Then we create an instance of `flow_graph_maker` (`wfm_rcv_graph`) with `vbox` as an argument passed to it (also with `frame` and the `panel` itself). In `wfm_rcv_graph.__init__()`, this `vbox` will append several spectrum analyzers or oscillograph (`wx.Window` objects) to the sizer by using `vbox.Add()`. Then the panel uses the sizer `vbox` position and size all these child-windows. Finally, we start the flow graph: `self.fg.start()`, the corresponding data would be displayed on the screen dynamically.

Let's go back to the code of our FM receiver example.

```
    if 1:
        pre_demod, fft_win1 = \
                     fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                                        512, quad_rate)
        self.connect (src, pre_demod)
        vbox.Add (fft_win1, 1, wx.EXPAND)
```

and the definition of the `make_fft_sink_c()` method:

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

Everything is much clearer now, right? `panel` is passed to `make_fft_sink_c()` as the 'parent' of this fft sink (`block.win`, a `wx.Window` object). The return value of `block.win` is saved in `fft_win1` and then appended to `vbox`.

`make_fft_sink_c()` takes seven parameters. `fft_size` is the number of samples used to perform FFT. `input_rate` is the sample frequency. `ymin` and `ymax` give the vertical range of the plot you wish to display on the screen.

Note that there is a complicated story behind the class `fft_sink_c`. We didn't talk about how fast Fourier transform is performed and how it is used as a block, but focusing on its interface to Python and wxPython. In fact, another Python package called 'Numeric' helps a lot here. However, we don't need to know all the details. Understanding how it interacts with wxPython and other blocks at the Python level would be sufficient.

2.3 The 'Oscillograph'- scope_sink

Another important GUI tool in GNU Radio is the 'soft oscillograph' - `scope_sink`. It's not used in our example, but it would be very helpful if you wish to see the waveforms in the time domain. Its usage is quite similar to the `fft_sink`:

```
if 1:
    scope_input, scope_win1 = \
                 scopesink.make_scope_sink_f (self, panel, "Title", self.fs)
    self.connect (signal, scope_input)
    vbox.Add (scope_win1, 1, wx.EXPAND)
```

Note that here `signal` should be a real float signal. If you wish to display a complex signal with I/Q channels, `make_scope_sink_c()` is the right choice. Copy these lines wherever you think a scope should appear, then connect it to the signal as a block. Refer to `/site-packages/gnuradio/wxgui/scopesink.py` for more details.

3 Handling command line arguments

Python fully supports creating programs that can be run on the command line, complete with command-line arguments and either short- or long- style flags to specify various options. Remember when we create an instance of `wfm_rcv_graph` in `stdpanel.__init__()`, we use:

```
self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
```

Each command line argument passed to the program will be saved in `sys.argv`, which is just a 'list'. In this list, `sys.argv[0]` is just the command itself (`wfm_rcv_gui.py` in our example). So actually all the arguments are saved in `sys.argv[1:]`. That explains why we use `IF_freq = parseargs(argv[1:])` to process the arguments.

You may want to use short- or long- style flags to add various options like '`-v`', or '`--help`'. Then the **optparse** module is exactly what you would like to use. **optparse** is a powerful and flexible command line interpreter. You may see this page to study it. Another example, located at

```
gnuradio-examples/python/usrp/fsk_r(t)x.py
```

gives a very good demonstration on how to use this parser.

4 conclusion

This tutorial completes our discussion on the FM receiver example. We mainly talk about how wxPython plays its role in GNU Radio. It might be a little bit involved to understand how those classes are organized together, but it won't be that difficult if we are patient enough. Besides, the good news is we can simple use those codes as templates, without worrying too much about the implementation details.

References

[1] **Python on-line tutorials**, http://www.python.org/doc/current/tut/

[2] **Python Library Reference - optparse**, http://www.python.org/doc/2.3/lib/module-optparse.html

[3] **wxPython on-line tutorials**, http://wxpython.org/tutorial.php

[4] **wxPython wiki, Getting started**, http://wiki.wxpython.org/index.cgi/Getting_20Started

Tutorial 8: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line - Part II

Dawei Shen[1]

July 13, 2005

Abstract

Let's continue our discussion on the FM example `wfm_rcv_gui.py`. The usage of the GUI tools in GNU Radio, which are built upon wxPython, will be shown. We will also introduce some useful programming tips on argument parsing. If you are interested in using or even developing the GUI tools, this article would be helpful.

Contents

1 Overview

In this article, we will complete our discussion on the FM receiver code `wfm_rcv_gui.py`. One exciting feature of GNU Radio, is it incorporates powerful GUI tools for displaying and analyzing the signal, emulating the real spectrum analyzer and the oscillograph. We will introduce the usage of the GUI tools, which are built upon wxPython. Next we will talk about how to handle the command line arguments in Python.

2 GUI tools in GNU Radio

The most intuitive and straightforward way to analyze a signal is to display it graphically, both in time domain and frequency domain. For the applications in the real world, we have the spectrum analyzer and the oscillograph to facilitate us. Fortunately, in the software radio world, we also have such nice tools, thanks to wxPython, which provides a flexible way to construct GUI tools.

2.1 The `Spectrum Analyzer' - fft_sink

Let's continue to read the code:

```
    if 1:
        pre_demod, fft_win1 = \
                    fftsink.make_fft_sink_c (self, panel, "Pre-
Demodulation",
                                                512, quad_rate)
        self.connect (src, pre_demod)
        vbox.Add (fft_win1, 1, wx.EXPAND)
```

This is the `soft spectrum analyzer', based on fast Fourier transformation (FFT) of the digital sequence. This `soft spectrum analyzer' is used as the signal sink. That's why it is named as `fftsink'. It's defined in the module wxgui.fftsink.py. The function make_fft_sink_c() serves as the public interface to create an instance of the fft sink:

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c
(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                    sample_rate=input_rate, y_per_div=
(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

First, we should point out that in Python, a function could return multiple values. make_fft_sink_c() returns two values: block is an instance of the class fft_sink_c, defined in the same module wxgui.fftsink.py. Another special feature of Python needs to be emphasized: Python supports multiple inheritance. fft_sink_c is derived from two classes: gr.hier_block and fft_sink_base. Being a `son' class of gr.hier_block implies that fft_sink_c can be treated as a normal block, which can be placed and connected in a flow graph, as the next line shows:

```
     self.connect (src, pre_demod)
```

block.win is obviously an attribute of block. In the definition of the class fft_sink_c, we can find its data type is the class fft_window, a subclass of wx.Window, also defined in the module `wxgui.fftsink.py`. We can think of it as a window that is going to be hang up on your screen. This window `block.win` will be used as the argument of the method `vbox.Add`.

2.2 How wxPython plays its role

To understand the other parts thoroughly, we need to know a little bit about wxPython, a GUI toolkit for Python. Interested readers may visit wxPython's website or tutorials' page for more information. It might be time consuming to explore all the technical details about wxPython. The good news is in GNU Radio, we can use the `spectrum analyzer` and `oscillograph` pretty much as it is. Just copy those lines anywhere you want with only a few changes.

Let's invest some time in wxPython's organism. The first thing to do is certainly to import all wxPython's components into current workspace, like the line `` `import wx`' `` does. Every wxPython application needs to derive a class from `wx.App` and provide an `OnInit()` method for it. The system calls this method as part of its startup/initialization sequence, in `wx.App.__init()__`. The primary purpose of `OnInit()` is to create the frames, windows, etc. necessary for the program to begin operation. After defining such a class, we need to instantiate an object of this class and start the application by calling its `MainLoop()` method, whose role is to handle the events. In our FM receiver example, where is such a class defined? Let's look at the last three lines:

```
if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

In fact, such a class, called `stdapp` has been created when we import the `stdgui` module.

```
     from gnuradio.wxgui import stdgui, fftsink
```

The final two lines again will probably be the same for all your wxPython applications. We simply create an instance of our application class, and then call its `MainLoop()` method. `MainLoop()` is the heart of the application and is where events are processed and dispatched to the various windows in the application. There are some tricks behind the scene. Don't worry about that.

Let's look at the definition of `stdapp` in `/gnuradio/wxgui/stugui.py`:

```
class stdapp (wx.App):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        self.flow_graph_maker = flow_graph_maker
        self.title = title
        # All our initialization must come before calling wx.App.
__init__.
```

```
        # OnInit is called from somewhere in the guts of __init__.
        wx.App.__init__ (self)

    def OnInit (self):
        frame = stdframe (self.flow_graph_maker, self.title)
        frame.Show (True)
        self.SetTopWindow (frame)
        return True
```

`stdapp` is right the class derived from `wx.App`. Its initialization method `__init__()` takes two arguments: `flow_graph_maker`, a class belonging to the flow graph family (remember the biggest class `wfm_rx_graph` we created is derived from `gr.flow_graph`?); `title`, the title of the whole application (`WFM RX` in our example). In `OnInit()` method, these two arguments are further used to create the object of `stdframe`.

```
class stdframe (wx.Frame):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        # print "stdframe.__init__"
        wx.Frame.__init__(self, None, -1, title)

        self.CreateStatusBar (2)
        mainmenu = wx.MenuBar ()

        menu = wx.Menu ()
        item = menu.Append (200, 'E&xit', 'Exit')
        self.Bind (wx.EVT_MENU, self.OnCloseWindow, item)
        mainmenu.Append (menu, "&File")
        self.SetMenuBar (mainmenu)

        self.Bind (wx.EVT_CLOSE, self.OnCloseWindow)
        self.panel = stdpanel (self, self, flow_graph_maker)
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(vbox)
        self.SetAutoLayout(True)
        vbox.Fit(self)

    def OnCloseWindow (self, event):
        self.flow_graph().stop()
        self.Destroy ()

    def flow_graph (self):
        return self.panel.fg
```

It's worth taking a while to understand the layout of a wxPython GUI. In wxPython, a `wx.Window` is anything which can take up visual space on the computer screen. Thus, the `wx.Window` class is the base class from which all visual elements are derived - including input fields, pull-down menus, buttons, etc. The `wx.Window` class defines all the behavior common to all visual GUI elements, including positioning, sizing, showing, giving focus, etc. If we're looking for an object to represent a window on the computer screen, don't look at `wx.Window`, look at `wx.Frame` instead. `wx.Frame` is derived from `wx.Window`. It implements all behavior specific to windows on the computer's screen. A `Frame' is a window whose size and position can (usually) be changed by the user. It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A `Frame' can contain any window that is not a frame or dialog. So to create a `window' on the computer screen, you create a `wx.Frame` (or one of its sub-classes, such as `wx.Dialog`), rather than a `wx.Window`.

Within a frame, you'll use a number of `wx.Window` sub-classes to flesh out the frame's contents, such as `wx.MenuBar`, `wx.StatusBar`, `wx.ToolBar`, sub-classes of `wx.Control` (eg. `wx.Button`, `wx.StaticText`, `wx.TextCtrl`, `wx.ComboBox`, etc.), or `wx.Panel`, which is a container to hold your various `wx.Control` objects. All visual elements (`wx.Window` objects and their subclasses) can hold sub-elements. A `wx.Frame` might hold a number of `wx.Panel` objects, which in turn hold a number of `wx.Button`, `wx.StaticText` and `wx.TextCtrl` objects.

In our example, `stdframe`, the subclass of `wx.Frame`, is used to create the `frame'. We make this frame appear by `showing' it using `frame.Show (True)`. The `SetTopWindow()` method simply tells that this frame is one of the main frames (in this case the only one) for the application. Notice the shape of the constructor of `wx.Frame`:

```
    wx.Frame(Parent, Id, "title")
```

Most of the constructors in wxPython have this shape: A parent object as a first parameter and an Id in a second parameter. As shown in the example, it's possible to use respectively None and -1 as default parameters, meaning the object has no parent and respectively a system-defined Id.

In `stdframe.__init__()`, we create a `panel` and place inside the frame.

```
    self.panel = stdpanel (self, self, flow_graph_maker)
```

The class `stdpanel` is derived from `wx.Panel`:

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        wx.Panel.__init__ (self, parent, -1)
        self.frame = frame
        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)

        self.fg.start ()
```

Note that `panel`'s parent is the `frame` object we create just now, meaning this panel is a subcomponent of the frame. The `frame` places the `panel` inside itself using a `wx.BoxSizer`, `vbox`. The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or nested hierarchies of either. A `wx.BoxSizer` will lay out its items in a simple row or column, depending on the orientation parameter passed to the constructor. In our example, `vbox = wx.BoxSizer(wx.VERTICAL)` tells the constructor all the items will be placed vertically. The `SetSizer()` call tells the frame which sizer to use. The call to `SetAutoLayout()` tells the frame to use the sizer to position and size your components. And finally, the call to `sizer.Fit()` tells the sizer to calculate the initial size and position for all its elements. If you are using sizers, this is the normal process you would go through to set up your window or frame's contents before it is displayed for the first time. The most important and useful method for a sizer is `add()`, it appends an item to the sizer. Its syntax looks like:

```
    Add(self, item, proportion=0, flag=0)
```

\``item`' is what you wish to append to the sizer, usually a `wx.Window` object. It can also be a child sizer. `proportion` is related to whether a child of a sizer can change its size in the main orientation of the wx.BoxSizer. There are several flags defined in wx, and they are used to determine how the sizer items behave and the border of the window. `wx.EXPAND` used in our example means the item will be expanded to fill the space allotted to the item. Refer to this page for a complete description.

Have you ever had this confusion: we define a \`big' class `wfm_rcv_graph`, but where do we use it? why do we never create an instance of this class? The secret is revealed in `stdpanel.__init__()`. The instance of `wfm_rcv_graph` is created here and the flow graph is started.

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        ......
        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)
        self.fg.start ()
```

We put a panel in the frame, but what do we put in the panel? We first create a new sizer `vbox` for the panel. Note that this `vbox` is different from the one defined in `stdframe`. Then we create an instance of `flow_graph_maker` (`wfm_rcv_graph`) with `vbox` as an argument passed to it (also with `frame` and the `panel` itself). In `wfm_rcv_graph.__init__()`, this `vbox` will append several spectrum analyzers or oscillograph (`wx.Window` objects) to the sizer by using `vbox.Add()`. Then the panel uses the sizer `vbox` position and size all these child-windows. Finally, we start the flow graph: `self.fg.start()`, the corresponding data would be displayed on the screen dynamically.

Let's go back to the code of our FM receiver example.

```
    if 1:
        pre_demod, fft_win1 = \
                    fftsink.make_fft_sink_c (self, panel, "Pre-
Demodulation",
                                                     512, quad_rate)
        self.connect (src, pre_demod)
        vbox.Add (fft_win1, 1, wx.EXPAND)
```

and the definition of the `make_fft_sink_c()` method:

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c
(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                sample_rate=input_rate, y_per_div=
(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

Everything is much clearer now, right? `panel` is passed to `make_fft_sink_c()` as the `parent' of this fft sink (`block.win`, a `wx.Window` object). The return value of `block.win` is saved in `fft_win1` and then appended to `vbox`.

`make_fft_sink_c()` takes seven parameters. `fft_size` is the number of samples used to perform FFT. `input_rate` is the sample frequency. `ymin` and `ymax` give the vertical range of the plot you wish to display on the screen.

Note that there is a complicated story behind the class `fft_sink_c`. We didn't talk about how fast Fourier transform is performed and how it is used as a block, but focusing on its interface to

Python and wxPython. In fact, another Python package called `Numeric' helps a lot here. However, we don't need to know all the details. Understanding how it interacts with wxPython and other blocks at the Python level would be sufficient.

2.3 The `Oscillograph'- scope_sink

Another important GUI tool in GNU Radio is the `soft oscillograph' - `scope_sink`. It's not used in our example, but it would be very helpful if you wish to see the waveforms in the time domain. Its usage is quite similar to the `fft_sink`:

```
if 1:
    scope_input, scope_win1 = \
                 scopesink.
make_scope_sink_f (self, panel, "Title", self.fs)
    self.connect (signal, scope_input)
    vbox.Add (scope_win1, 1, wx.EXPAND)
```

Note that here `signal` should be a real float signal. If you wish to display a complex signal with I/Q channels, `make_scope_sink_c()` is the right choice. Copy these lines wherever you think a scope should appear, then connect it to the signal as a block. Refer to `/site-packages/gnuradio/wxgui/scopesink.py` for more details.

3 Handling command line arguments

Python fully supports creating programs that can be run on the command line, complete with command-line arguments and either short- or long- style flags to specify various options. Remember when we create an instance of `wfm_rcv_graph` in `stdpanel.__init__()`, we use:

```
    self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
```

Each command line argument passed to the program will be saved in `sys.argv`, which is just a `list'. In this list, `sys.argv[0]` is just the command itself (`wfm_rcv_gui.py` in our example). So actually all the arguments are saved in `sys.argv[1:]`. That explains why we use `IF_freq = parseargs(argv[1:])` to process the arguments.

You may want to use short- or long- style flags to add various options like `` `-v' ``, or `` `--help' ``. Then the **optparse** module is exactly what you would like to use. **optparse** is a powerful and flexible command line interpreter. You may see this page to study it. Another example, located at

```
    gnuradio-examples/python/usrp/fsk_r(t)x.py
```

gives a very good demonstration on how to use this parser.

4 conclusion

This tutorial completes our discussion on the FM receiver example. We mainly talk about how wxPython plays its role in GNU Radio. It might be a little bit involved to understand how those classes are organized together, but it won't be that difficult if we are patient enough. Besides, the good news is we can simple use those codes as templates, without worrying too much about the implementation details.

References

[1]
Python on-line tutorials, http://www.python.org/doc/current/tut/

[2]
Python Library Reference - optparse, http://www.python.org/doc/2.3/lib/module-optparse.html

[3]
wxPython on-line tutorials, http://wxpython.org/tutorial.php

[4]
wxPython wiki, Getting started, http://wiki.wxpython.org/index.cgi/Getting_20Started

---

Footnotes:

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

---

File translated from T_EX by T_TH, version 3.68.

On 24 Jul 2005, 01:35.

Tutorial 9: A Dictionary of the GNU Radio blocks

Dawei Shen[*]

August 21, 2005

Abstract

A dictionary of the GNU Radio blocks. This article takes a tour around the most frequently used blocks, explaining the syntax and how to use them.

1 Introduction

When we use Matlab to do simulation, it is believed that in order to write the code cleanly and efficiently, we need to memorize a number of Matlab built-in functions and tool boxes well and use them skillfully. The same applies to GNU Radio. About 100 frequently used blocks come with GNU Radio. For a certain number of applications, we can complete the designing using these existing blocks, programming only on the Python level, without the need to write our own blocks. So in this article, we will take a tour around the GNU Radio blocks.

A good way to go through the blocks is to study the two GNU Radio documentations generated by Doxygen. They will be generated when you install `gnuradio-core` and `usrp` packages, assuming you have Doxygen installed. They are located at:

```
/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html
/usr/local/share/doc/usrp-x.xcvs/html/index.html
```

You may like to bookmark them in your web browser. The first one is also available on-line.

A block is actually a class implemented in C++. For example, in the FM receiver example, we use the block `gr.quadrature_demod_cf`. This block corresponds to the class `gr_quadrature_demod_cf` implemented in C++. The SWIG creates its interface to Python. The SWIG does some magic work so that from Python's point of view, the block becomes a class called `quadrature_demod_cf` defined in the module `gr`, so that we can access to the block using `gr.quadrature_demod_cf` in Python. When you look at the Doxygen documentations, the prefix such as `gr`, `qa`, `usrp` corresponds to the module name in Python and the part after the prefix is the real name of the block in that module, such as `quadrature_demod_cf`, `fir_filter_fff`. So if we talk about a block named '`A_B_C_...`', we will use it as '`A.B_C_...`' in Python.

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

2 Signal Sources

2.1 Sinusoidal and constant sources

Block: `gr.sig_source_X`.

Usage:

```
gr.sig_source_c [f, i, s]   ( double sampling_freq,
                              gr_waveform_t waveform,
                              double frequency,
                              double amplitude,
                              gr_complex [float, integer, short] offset )
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer) or `s` (2 byte short integer). The `offset` argument is the same type as the signal. The `waveform` argument indicates the type of the wave form. `gr_waveform_t` is an enumeration type defined in `gr_sig_source_waveform.h`. Its value can be:

```
gr.GR_CONST_WAVE
gr.GR_COS_WAVE
gr.GR_SIN_WAVE
```

When you use `gr.GR_CONST_WAVE`, the output will be a constant voltage, which is the amplitude plus the offset.

2.2 Noise sources

Block: `gr.noise_source_X`.

Usage:

```
gr.noise_source_c [f, i, s] ( gr_noise_type_t type,
                              float amplitude,
                              long seed )
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer) or `s` (2 byte short integer). The `type` argument indicates the type of the noise. `gr_noise_type_t` is an enumeration type defined in `gr_noise_type.h`. Its value can be:

```
GR_UNIFORM
GR_GAUSSIAN
GR_LAPLACIAN
GR_IMPULSE
```

Choosing `GR_UNIFORM` generates uniformly distributed signal between `[-amplitude, amplitude]`. `GR_GAUSSIAN` gives us normally distributed deviate with zero mean and variance $amplitude^2$. Similarly, `GR_LAPLACIAN` and `GR_IMPLUSE` represent a Laplacian distribution and a impulse distribution respectively. All these noise source blocks are based on the pseudo random number generator block gr.random. You may take a look at `/gnuradio-core/src/lib/general/gr_random.h(cc)` to see how to generate a random number following various distributions.

2.3 Null sources

Block: `gr.null_source`.

Usage:

```
gr.null_source ( size_t  sizeof_stream_item )
```

Notes: `gr.null_source` produces constant zeros. The argument `sizeof_stream_item` specifies the data type of the zero stream, such as `gr_complex`, `float`, `integer` and so on.

2.4 Vector sources

Block: `gr.vector_source_X`.

Usage:

```
gr.vector_source_c [f, i, s, b] ( const std::vector< gr_complex > & data,
                                  bool    repeat = false )

(gr_complex can be replaced by float, integer, short, unsigned char)
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer), `s` (2 byte short integer), or `b` (1 byte unsigned char). These sources get their data from a vector. The argument `repeat` decides whether the data in the vector is sent repeatedly. As an example, we can use the block in this way in Python:

```
        src_data = (-3, 4, -5.5, 2, 3)
        src = gr.vector_source_f (src_data)
```

2.5 File sources

Block: `gr.file_source`

Usage:

```
gr.file_source ( size_t         itemsize,
                 const char *   filename,
                 bool           repeat )
```

Notes: `gr.file_source` reads the data stream from a file. The name of the file is specified by `filename`. The first argument `itemsize` determines the data type of the stream, such as `gr_complex`, `float`, `unsigned char`. The argument `repeat` decides whether the data in the file is sent repeatedly. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/payload.dat", TRUE)
```

2.6 Audio source

Block: `gr.audio_source`

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: `audio_source` reads data from the audio line-in. The argument `sampling_rate` specifies the data rate of the source, in samples per second.

2.7 USRP source

Block: `usrp.source_c [s]`

Usage:

```
usrp.source_c (s) (int              which_board,
                   unsigned int     decim_rate,
                   int              nchan = 1,
                   int              mux = -1,
                   int              mode = 0 )
```

Notes: when you design a receiver using GNU Radio, i.e. working on the RX path, probably you need the USRP as the data source. The suffix `c` (complex), or `s` (short) specifies the data type of the stream from USRP. Most likely the complex source (I/Q quadrature from the digital down converter (DDC)) would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `decim_rate` tells the digital down converter (DDC) the decimation factor D. `nchan` specifies the number of channels, 1, 2 or 4. `mux` sets input MUX configuration, which determines which ADC (or constant zero) is connected to each DDC input (see tutorial 4 for details). '-1' means we preserve the default settings. `mode` sets the FPGA mode, which we seldom touch. The default value is 0, representing the normal mode. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_decim = 250
src = usrp.source_c (0, usrp_decim)
```

3 Signal Sinks

3.1 Null sinks

Block: `gr.null_sink`.

Usage:

```
gr.null_sink ( size_t  sizeof_stream_item )
```

Notes: `gr.null_sink` does nothing but eat up your stream. The argument `sizeof_stream_item` specifies the data type of the zero stream, such as `gr_complex`, `float`, `integer` and so on.

3.2 Vector sinks

Block: `gr.vector_sink_X`.

Usage:

```
gr.vector_sink_c [f, i, s, b] ()
```

Notes: The suffix `X` indicates the data type of the signal sink. It can be `c` (complex), `f` (float), `i` (4 byte integer), `s` (2 byte short integer), or `b` (1 byte unsigned char). These sinks write the data into a vector. As an example, we can use the block in this way in Python:

```
dst = gr.vector_sink_f ()
```

3.3 File sinks

Block: `gr.file_sink`

Usage:

```
gr.file_sink ( size_t         itemsize,
               const char *   filename )
```

Notes: `gr.file_source` writes the data stream to a file. The name of the file is specified by `filename`. The first argument `itemsize` determines the data type of the input stream, such as `gr_complex`, `float`, `unsigned char`. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/rx.dat")
```

3.4 Audio sink

Block: `gr.audio_sink`

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: When you finish the signal processing and wish to play the signal through the speaker, you should use the audio sink. `audio_sink` will output the data to the sound card at the sampling rate of `sampling_rate`.

3.5 USRP sink

Block: `usrp.sink_c [s]`

Usage:

```
usrp.sink_c (s) ( int               which_board,
                  unsigned int      interp_rate,
                  int               nchan = 1,
                  int               mux = -1 )
```

Notes: when you design a transmitter using GNU Radio, i.e. working on the TX path, probably you need the USRP as the data sink. The suffix `c` (complex), or `s` (short) specifies the data type of the stream going into USRP. Most likely the complex sink would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `interp_rate` tells the interpolator on the FPGA the interpolation factor I. Note that the interpolation rate I must be in the range `[4, 512]`, and must be a multiple of 4. `nchan` specifies the number of channels, 1 or 2. `mux` sets output MUX configuration, which determines which DAC is connected to each interpolator output (or disabled). '-1' means we preserve the default settings. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_interp = 256
src = usrp.sink_c (0, usrp_interp)
```

4 Simple Operators

4.1 Adding a constant

Block: `gr.add_const_XX`

Usage:

```
gr.add_const_cc [ff, ii, ss, sf] ( gr_complex [float, integer, short] k )
```

Notes: The suffix `XX` contains two characters. The first one indicates the data type of the input stream while the second one tells the type of the output stream. It can be `cc` (complex to complex), `ff` (float to float), `ii` (4 byte integer to integer), `ss` (2 byte short integer to short integer) or `sf` (short integer to float). The `gr.add_const_XX` block adds a constant to the input stream so that the signal has a different offset. Note that for `gr.add_const_sf`, the argument k is float. We add a float constant to a short integer stream, and the output stream becomes float.

4.2 Adder

Block: `gr.add_XX`

Usage:

```
gr.add_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.add_XX` adds all input streams together. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
adder = gr.add_cc ()
fg.connect (stream1, (adder, 0))
fg.connect (stream2, (adder, 1))
fg.connect (stream3, (adder, 2))
fg.connect (adder, outputstream)
```

Then the output of the `adder` is `stream1+stream2+stream3`.

4.3 Subtractor

Block: `gr.sub_XX`

Usage:

```
gr.sub_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. gr.sub_XX subtracts across all input streams. output = input_0 - input_1 - input_2.... The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
subtractor = gr.sub_cc ()
fg.connect (stream1, (subtractor, 0))
fg.connect (stream2, (subtractor, 1))
fg.connect (stream3, (subtractor, 2))
fg.connect (subtractor, outputstream)
```

Then the output of the subtractor is stream1-stream2-stream3.

4.4 Multiplying a constant

Block: gr.multiply_const_XX

Usage:

```
gr.add_const_cc [ff, ii, ss] ( gr_complex [float, integer, short] k )
```

Notes: The suffix XX indicates the data type of the input and output streams. The gr.multiply_const_XX block multiplies the input stream by a constant k.

4.5 Multiplier

Block: gr.multiply_XX

Usage:

```
gr.multiply_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. gr.multiply_XX computes the product of all input streams. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
multiplier = gr.multiply_cc ()
fg.connect (stream1, (multiplier, 0))
fg.connect (stream2, (multiplier, 1))
fg.connect (stream3, (multiplier, 2))
fg.connect (multiplier, outputstream)
```

Then the output of the multiplier is stream1×stream2×stream3.

4.6 Divider

Block: gr.divide_XX

Usage:

```
gr.divide_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.divide_XX` divides across all input streams. `output = input_0 / input_1 / input_2...`. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
divider = gr.divide_cc ()
fg.connect (stream1, (divider, 0))
fg.connect (stream2, (divider, 1))
fg.connect (stream3, (divider, 2))
fg.connect (divider, outputstream)
```

Then the output of the `subtractor` is stream1÷stream2÷stream3.

4.7 Log function

Block: `gr.nlog10_ff`

Usage:

```
gr.nlog10_ff ( float n,
               unsigned vlen )
```

Notes: `gr.nlog10_ff` computes $n \times \log_{10}(input)$. Input and output are both float numbers. Ignore the argument `vlen`, the vector length, using its default value 1.

5 Type Conversions

5.1 Complex Conversions

Block:

```
gr.complex_to_float
gr.complex_to_real
gr.complex_to_imag
gr.complex_to_mag
gr.complex_to_arg
```

Usage:

```
gr.complex_to_float( unsigned int vlen )
gr.complex_to_real( unsigned int vlen )
gr.complex_to_imag( unsigned int vlen )
gr.complex_to_mag( unsigned int vlen )
gr.complex_to_arg( unsigned int vlen )
```

Notes: The argument `vlen` is the vector length, we almost always use the default value 1. So just ignore the argument. These blocks convert a complex signal to separate real & imaginary float streams, just the real part, just the imaginary part, the magnitude of the complex signal and the phase angle of the complex signal. Note that the block `gr.complex_to_float` could have 1 or 2 outputs. If it is connected to only one output, then the output is the real part of the complex signal. Its effect is equivalent to `gr.complex_to_real`.

5.2 Float Conversions

Block:

```
gr.float_to_complex
gr.float_to_short
gr.short_to_float
```

Usage:

```
gr.float_to_complex ( )
gr.float_to_short ( )
gr.short_to_float ( )
```

Notes: These blocks are like 'adapters', used to provide the interface between two blocks with different types. Note that the block `gr.float_to_complex` may have 1 or 2 inputs. If there is only one, then the input signal is the real part of the output signal, while the imaginary part is constant 0. If there are two inputs, they serve as the real and imaginary parts of the output signal respectively.

6 Filters

6.1 FIR Designer

Block: `gr.firdes`

Notes: `gr.firdes` has several static public member functions used to design different types of FIR filters. These functions return a vector containing the FIR coefficients. The returned vector is often used as an argument of other FIR filter blocks.

6.1.1 Low Pass Filter

Usage:

```
vector< float > gr.firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76)    [static]
```

Notes: `low_pass` is a static public member function in the class `gr.firdes`. It designs the FIR filter coefficients (`taps`) given the filter specifications. Here the argument `window` is the type of the window used in the FIR filter design. Valid values include

```
WIN_HAMMING
WIN_HANN
WIN_BLACKMAN
WIN_RECTANGULAR
```

6.1.2 High Pass Filter

Usage:

```
vector< float > gr.firdes::high_pass ( double      gain,
                                       double      sampling_freq,
                                       double      cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.3 Band Pass Filter

Usage:

```
vector< float > gr.firdes::band_pass ( double      gain,
                                       double      sampling_freq,
                                       double      low_cutoff_freq,
                                       double      high_cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.4 Band Reject Filter

Usage:

```
vector< float > gr.firdes::band_reject ( double      gain,
                                         double      sampling_freq,
                                         double      low_cutoff_freq,
                                         double      high_cutoff_freq,
                                         double      transition_width,
                                         win_type    window = WIN_HAMMING,
                                         double      beta = 6.76)   [static]
```

6.1.5 Hilbert Filter

Usage:

```
vector< float > gr.firdes::hilbert ( unsigned int     ntaps,
                                     win_type    windowtype = WIN_RECTANGULAR,
                                     double      beta = 6.76 )   [static]
```

Notes: `gr.firdes::hilbert` designs a Hilbert transform filter. `ntaps` is the number of taps, which must be odd.

6.1.6 Raised Cosine Filter

Usage:

```
vector< float > gr.firdes::root_raised_cosine ( double     gain,
                                                double     sampling_freq,
                                                double     symbol_rate,
                                                double     alpha,
                                                int        ntaps )   [static]
```

Notes: `gr.firdes::root_raised_cosine` designs a root cosine FIR filter. The argument `alpha` is the excess bandwidth factor. `ntaps` is the number of taps.

6.1.7 Gaussian Filter

Usage:

```
vector< float > gr.firdes::gaussian ( double     gain,
                                      double     sampling_freq,
                                      double     symbol_rate,
                                      double     bt,
                                      int        ntaps )   [static]
```

Notes: `gr.firdes::gaussian` designs a Gaussian filter. The argument `ntaps` is the number of taps.

6.2 FIR Decimation Filters

Block:

```
gr.fir_filter_ccc
gr.fir_filter_ccf
gr.fir_filter_fcc
gr.fir_filter_fff
gr.fir_filter_fsf
gr.fir_filter_scc
```

Usage:

```
gr.fir_filter_ccc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_ccf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fcc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_fff (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fsf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_scc (int decimation,
                   const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the decimation rate of the FIR filter. If it is 1, then it's just a normal 1:1 FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.3 FIR Interpolation Filters

Block:

```
gr.interp_fir_filter_ccc
gr.interp_fir_filter_ccf
gr.interp_fir_filter_fcc
gr.interp_fir_filter_fff
gr.interp_fir_filter_fsf
gr.interp_fir_filter_scc
```

Usage:

```
gr.interp_fir_filter_ccc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
gr.interp_fir_filter_ccf (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_fcc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
gr.interp_fir_filter_fff (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_fsf (unsigned interpolation,
                          const std::vector< float > & taps )
gr.interp_fir_filter_scc (unsigned interpolation,
                          const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the interpolation rate of the FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.4 Digital Down Converter with FIR Decimation Filters

Block:

```
gr.freq_xlating_fir_filter_ccc
gr.freq_xlating_fir_filter_ccf
gr.freq_xlating_fir_filter_fcc
gr.freq_xlating_fir_filter_fcf
gr.freq_xlating_fir_filter_scc
gr.freq_xlating_fir_filter_scf
```

Usage:

```
gr.freq_xlating_fir_filter_ccc [ccf, fcc, fcf, scc, scf]
```

```
                                          ( int      decimation,
                                            const std::vector< gr_complex [float] > &   taps,
                                            double      center_freq,
                                            double      sampling_freq )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

These blocks are used as FIR filters combined with frequency translation. Recall that in tutorial 4, in the FPGA there are digital down converters (DDC) translating the signal from IF to baseband. Then the following decimator downsamples the signal and selects a narrower band of the signal by low-pass filtering it. These blocks do the same things except in software. These classes efficiently combine a frequency translation (typically down conversion) with an FIR filter (typically low-pass) and decimation. It is ideally suited for a 'channel selection filter' and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

6.5 Hilbert Transform Filter

Block: `gr.hilbert_fc`

Usage:

```
gr.hilbert_fc( unsigned int ntaps )
```

Notes: `gr.hilbert_fc` is a Hilbert transformer. The real output is input appropriately delayed. Imaginary output is Hilbert filtered (90 degree phase shift) version of input. We only need to specify the number of taps `ntaps` when using the block. The Hilbert filter designer `gr.firdes::hilbert` is used implicitly in the implementation of the block `gr.hilbert_fc`.

6.6 Filter-Delay Combination Filter

Block: `gr.filter_delay_fc`

Usage:

```
gr.filter_delay_fc( const std::vector< float > &  taps )
```

Notes: This is a filter-delay combination block. The block takes one or two float stream and outputs a complex stream. If only one float stream is input, the real output is a delayed version of this input and the imaginary output is the filtered output. If two floats are connected to the input, then the real output is the delayed version of the first input, and the imaginary output is the filtered output. The delay in the real path accounts for the group delay introduced by the filter in the imaginary path. The filter taps needs to be calculated using `gr.firdes` before initializing this block.

6.7 IIR Filter

Block: `gr.iir_filter_ffd`

Usage:

```
gr.iir_filter_ffd ( const std::vector< double > &  fftaps,
                    const std::vector< double > &  fbtaps)
```

Notes: The suffix `ffd` means float input, float output and double taps. This IIR filter uses the Direct Form I implementation, where `fftaps` contains the feed-forward taps, and `fbtaps` the feedback ones. `fftaps` and `fbtaps` must have equal numbers of taps. The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

Note that some texts define the system function with a '+' in the denominator. If you're using that convention, you'll need to negate the feedback taps.

6.8 Single Pole IIR Filter

Block: `gr.single_pole_iir_filter_ff`

Usage:

```
gr.single_pole_iir_filter_ff ( double alpha,
                               unsigned int    vlen )
```

Notes: This is a single pole IIR filter with float input, float output. The input and output satisfy a difference equation of the form:

$$y[n] - (1-\alpha)y[n-1] = \alpha x[n]$$

with the corresponding rational system function

$$H(z) = \frac{\alpha}{1-(1-\alpha)z^{-1}}$$

Note that some texts define the system function with a + in the denominator. If you're using that convention, you'll need to negate the feedback tap. The argument `vlen` is the vector length. We usually use its default value 1, so just ignore it.

7 FFT

Block:

```
gr.fft_vcc
gr.fft_vfc
```

Usage:

```
gr.fft_vcc ( int      fft_size,
             bool     forward,
             bool     window )
gr.fft_vfc ( int      fft_size,
             bool     forward,
             bool     window )
```

Notes: These blocks are used to compute the fast Fourier transforms of the input sequence. For `gr.fft_vcc`, it computes forward or reverse FFT, with complex vector in and complex vector out. For `gr.fft_vfc` it computes forward FFT with float vector in and complex vector out.

8 Other useful blocks

8.1 FM Modulation and Demodulation

Block:

```
gr.frequency_modulator_fc
gr.quadrature_demod_cf
```

Usage:

```
gr.frequency_modulator_fc ( double sensitivity )
gr.quadrature_demod_cf ( float gain )
```

Notes: The `gr.frequency_modulator_fc` block is the FM modulator. The instantaneous frequency of the output complex signal is proportional to the input float signal. We have seen `gr.quadrature_demod_cf` in the FM receiver example. It calculates the instantaneous frequency of the input signal.

8.2 Numerically Controlled Oscillator

Block: `gr.fxpt_nco`

Usage: This block is used to generate sinusoidal waves. We can set or adjust the phase and frequency of the oscillator by several public member functions of the class. The functions include:

```
void    set_phase (float angle)
void    adjust_phase (float delta_phase)
void    set_freq (float angle_rate)
void    adjust_freq (float delta_angle_rate)
void    step ()
void    step (int n)
float   get_phase () const
float   get_freq () const
void    sincos (float *sinx, float *cosx) const
float   cos () const
float   sin () const
```

We use `cos()` or `sin()` function to get the sinusoidal samples. They calculate sin and cos values according to the current phase. The `freq` is actually the phase difference between consecutive samples. When we call `step()` method, the current phase will increase by the `freq`.

8.3 Blocks for digital transmission

Block:

```
gr.bytes_to_syms
gr.simple_framer
gr.simple_correlator
```

Usage:

```
gr.bytes_to_syms ( )
gr.simple_framer ( int payload_bytesize )
gr.simple_correlator ( int payload_bytesize )
```

Notes: `gr.bytes_to_syms ( )` converts a byte sequence (for example, a unsigned char stream) to a ± sequence, i.e. the digital binary symbols. `gr.simple_framer` packs a byte stream into packets, with the length of `payload_bytesize`. Then necessary synchronization and command bytes are added at the head. `gr.simple_correlator` is a digital detector, which synchronizes the symbol and frame, finally detects the digital information correctly. The real implementation of these blocks is a little bit complicated. Please study the FSK example `fsk_r[t]x.py` and the source code of these blocks. These blocks are very helpful when we want to design digital transmission schemes.

9 Wrap up

This tutorial reviews some of the most frequently used blocks in GNU Radio. Using these blocks skillfully would be very important and useful. Certainly we only discuss a fraction of the available blocks. There are some other more advanced, less used blocks that we didn't talk about. Also more and more blocks are coming as GNU Radio gets more popular (maybe including your block one day). Our introduction is very simple. If you wish to know all the details about a block, please go to its documentation page and then read its source code directly. The source code is the best place to understand what's going on in a block thoroughly. Studying some of the examples to see how a block a used is also a very good way.

References

[1] **GNU Radio 2.x Documentation** http://www.gnu.org/software/gnuradio/doc/index.html

Tutorial 9: A Dictionary of the GNU Radio blocks

Dawei Shen[1]

August 21, 2005

Abstract

A dictionary of the GNU Radio blocks. This article takes a tour around the most frequently used blocks, explaining the syntax and how to use them.

Contents

1 Introduction

When we use Matlab to do simulation, it is believed that in order to write the code cleanly and efficiently, we need to memorize a number of Matlab built-in functions and tool boxes well and use them skillfully. The same applies to GNU Radio. About 100 frequently used blocks come with GNU Radio. For a certain number of applications, we can complete the designing using these existing blocks, programming only on the Python level, without the need to write our own blocks. So in this article, we will take a tour around the GNU Radio blocks.

A good way to go through the blocks is to study the two GNU Radio documentations generated by Doxygen. They will be generated when you install `gnuradio-core` and `usrp` packages, assuming you have Doxygen installed. They are located at:

```
/usr/local/share/doc/gnuradio-core-x.xcvs/html/index.html
/usr/local/share/doc/usrp-x.xcvs/html/index.html
```

You may like to bookmark them in your web browser. The first one is also available on-line.

A block is actually a class implemented in C++. For example, in the FM receiver example, we use the block `gr.quadrature_demod_cf`. This block corresponds to the class `gr_quadrature_demod_cf` implemented in C++. The SWIG creates its interface to Python. The SWIG does some magic work so that from Python's point of view, the block becomes a class called `quadrature_demod_cf` defined in the module `gr`, so that we can access to the block using `gr.quadrature_demod_cf` in Python. When you look at the Doxygen documentations, the prefix such as `gr`, `qa`, `usrp` corresponds to the module name in Python and the part after the prefix is the real name of the block in that module, such as `quadrature_demod_cf`, `fir_filter_fff`. So if we talk about a block named `` `A_B_C_...' ``, we will use it as `` `A.B_C_...' `` in Python.

2 Signal Sources

2.1 Sinusoidal and constant sources

Block: gr.sig_source_X.

Usage:

```
gr.sig_source_c [f, i, s]   ( double sampling_freq,
                              gr_waveform_t waveform,
                              double frequency,
                              double amplitude,
                              gr_complex [float, integer, short] offset )
```

Notes: The suffix X indicates the data type of the signal source. It can be c (complex), f (float), i (4 byte integer) or s (2 byte short integer). The offset argument is the same type as the signal. The waveform argument indicates the type of the wave form. gr_waveform_t is an enumeration type defined in gr_sig_source_waveform.h. Its value can be:

```
gr.GR_CONST_WAVE
gr.GR_COS_WAVE
gr.GR_SIN_WAVE
```

When you use gr.GR_CONST_WAVE, the output will be a constant voltage, which is the amplitude plus the offset.

2.2 Noise sources

Block: gr.noise_source_X.

Usage:

```
gr.noise_source_c [f, i, s] ( gr_noise_type_t type,
                              float amplitude,
                              long seed )
```

Notes: The suffix X indicates the data type of the signal source. It can be c (complex), f (float), i (4 byte integer) or s (2 byte short integer). The type argument indicates the type of the noise. gr_noise_type_t is an enumeration type defined in gr_noise_type.h. Its value can be:

```
GR_UNIFORM
GR_GAUSSIAN
GR_LAPLACIAN
GR_IMPULSE
```

Choosing GR_UNIFORM generates uniformly distributed signal between [-amplitude, amplitude]. GR_GAUSSIAN gives us normally distributed deviate with zero mean and variance amplitude^2. Similarly, GR_LAPLACIAN and GR_IMPLUSE represent a Laplacian distribution and a impulse distribution respectively. All these noise source blocks are based on the pseudo random number generator block gr.random. You may take a look at /gnuradio-core/src/lib/general/gr_random.h(cc) to see how to generate a random number following various distributions.

2.3 Null sources

Block: `gr.null_source`.

Usage:

```
gr.null_source ( size_t  sizeof_stream_item )
```

Notes: `gr.null_source` produces constant zeros. The argument `sizeof_stream_item` specifies the data type of the zero stream, such as `gr_complex`, `float`, `integer` and so on.

2.4 Vector sources

Block: `gr.vector_source_X`.

Usage:

```
gr.vector_source_c [f, i, s, b] ( const std::vector< gr_complex > & data,
                                  bool    repeat = false )

(gr_complex can be replaced by float, integer, short, unsigned char)
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer), `s` (2 byte short integer), or `b` (1 byte unsigned char). These sources get their data from a vector. The argument `repeat` decides whether the data in the vector is sent repeatedly. As an example, we can use the block in this way in Python:

```
        src_data = (-3, 4, -5.5, 2, 3)
        src = gr.vector_source_f (src_data)
```

2.5 File sources

Block: `gr.file_source`

Usage:

```
gr.file_source ( size_t         itemsize,
                 const char *   filename,
                 bool           repeat )
```

Notes: `gr.file_source` reads the data stream from a file. The name of the file is specified by `filename`. The first argument `itemsize` determines the data type of the stream, such as `gr_complex`, `float`, `unsigned char`. The argument `repeat` decides whether the data in the file is sent repeatedly. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/payload.dat", TRUE)
```

2.6 Audio source

Block: `gr.audio_source`

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: `audio_source` reads data from the audio line-in. The argument `sampling_rate` specifies the data rate of the source, in samples per second.

2.7 USRP source

Block: `usrp.source_c [s]`

Usage:

```
usrp.source_c (s) (int               which_board,
                   unsigned int      decim_rate,
                   int               nchan = 1,
                   int               mux = -1,
                   int               mode = 0 )
```

Notes: when you design a receiver using GNU Radio, i.e. working on the RX path, probably you need the USRP as the data source. The suffix `c` (complex), or `s` (short) specifies the data type of the stream from USRP. Most likely the complex source (I/Q quadrature from the digital down converter (DDC)) would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `decim_rate` tells the digital down converter (DDC) the decimation factor D. `nchan` specifies the number of channels, 1, 2 or 4. `mux` sets input MUX configuration, which determines which ADC (or constant zero) is connected to each DDC input (see tutorial 4 for details). `-1' means we preserve the default settings. `mode` sets the FPGA mode, which we seldom touch. The default value is 0, representing the normal mode. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_decim = 250
src = usrp.source_c (0, usrp_decim)
```

3 Signal Sinks

3.1 Null sinks

Block: `gr.null_sink`.

Usage:

```
gr.null_sink ( size_t  sizeof_stream_item )
```

Notes: `gr.null_sink` does nothing but eat up your stream. The argument `sizeof_stream_item` specifies the data type of the zero stream, such as `gr_complex`, `float`, `integer` and so on.

3.2 Vector sinks

Block: `gr.vector_sink_X`.

Usage:

```
gr.vector_sink_c [f, i, s, b] ()
```

Notes: The suffix `X` indicates the data type of the signal sink. It can be `c` (complex), `f` (float), `i` (4 byte integer), `s` (2 byte short integer), or `b` (1 byte unsigned char). These sinks write the data into a vector. As an example, we can use the block in this way in Python:

```
dst = gr.vector_sink_f ()
```

3.3 File sinks

Block: `gr.file_sink`

Usage:

```
gr.file_sink ( size_t         itemsize,
               const char *   filename )
```

Notes: `gr.file_source` writes the data stream to a file. The name of the file is specified by `filename`. The first argument `itemsize` determines the data type of the input stream, such as `gr_complex`, `float`, `unsigned char`. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/rx.dat")
```

3.4 Audio sink

Block: `gr.audio_sink`

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: When you finish the signal processing and wish to play the signal through the speaker, you should use the audio sink. `audio_sink` will output the data to the sound card at the sampling rate of `sampling_rate`.

3.5 USRP sink

Block: usrp.sink_c [s]

Usage:

```
usrp.sink_c (s) ( int               which_board,
                  unsigned int      interp_rate,
                  int               nchan = 1,
                  int               mux = -1 )
```

Notes: when you design a transmitter using GNU Radio, i.e. working on the TX path, probably you need the USRP as the data sink. The suffix `c` (complex), or `s` (short) specifies the data type of the stream going into USRP. Most likely the complex sink would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `interp_rate` tells the interpolator on the FPGA the interpolation factor I. Note that the interpolation rate I must be in the range `[4, 512]`, and must be a multiple of 4. `nchan` specifies the number of channels, 1 or 2. `mux` sets output MUX configuration, which determines which DAC is connected to each interpolator output (or disabled). `-1' means we preserve the default settings. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_interp = 256
src = usrp.sink_c (0, usrp_interp)
```

4 Simple Operators

4.1 Adding a constant

Block: gr.add_const_XX

Usage:

```
gr.add_const_cc [ff, ii, ss, sf] ( gr_complex [float, integer, short] k )
```

Notes: The suffix `XX` contains two characters. The first one indicates the data type of the input stream while the second one tells the type of the output stream. It can be `cc` (complex to complex), `ff` (float to float), `ii` (4 byte integer to integer), `ss` (2 byte short integer to short integer) or `sf` (short integer to float). The `gr.add_const_XX` block adds a constant to the input stream so that the signal has a different offset. Note that for `gr.add_const_sf`, the argument k is float. We add a float constant to a short integer stream, and the output stream becomes float.

4.2 Adder

Block: gr.add_XX

Usage:

```
gr.add_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. `gr.add_XX` adds all input streams together. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
adder = gr.add_cc ()
fg.connect (stream1, (adder, 0))
fg.connect (stream2, (adder, 1))
fg.connect (stream3, (adder, 2))
fg.connect (adder, outputstream)
```

Then the output of the `adder` is `stream1+stream2+stream3`.

4.3 Subtractor

Block: `gr.sub_XX`

Usage:

```
gr.sub_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. `gr.sub_XX` subtracts across all input streams. `output = input_0 - input_1 - input_2...`. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
subtractor = gr.sub_cc ()
fg.connect (stream1, (subtractor, 0))
fg.connect (stream2, (subtractor, 1))
fg.connect (stream3, (subtractor, 2))
fg.connect (subtractor, outputstream)
```

Then the output of the `subtractor` is `stream1-stream2-stream3`.

4.4 Multiplying a constant

Block: `gr.multiply_const_XX`

Usage:

```
gr.add_const_cc [ff, ii, ss] ( gr_complex [float, integer, short] k )
```

Notes: The suffix XX indicates the data type of the input and output streams. The `gr.multiply_const_XX` block multiplies the input stream by a constant k.

4.5 Multiplier

Block: `gr.multiply_XX`

Usage:

```
gr.multiply_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.multiply_XX` computes the product of all input streams. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
multiplier = gr.multiply_cc ()
fg.connect (stream1, (multiplier, 0))
fg.connect (stream2, (multiplier, 1))
fg.connect (stream3, (multiplier, 2))
fg.connect (multiplier, outputstream)
```

Then the output of the `multiplier` is stream1×stream2×stream3.

4.6 Divider

Block: `gr.divide_XX`

Usage:

```
gr.divide_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.divide_XX` divides across all input streams. `output = input_0 / input_1 / input_2...`. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
divider = gr.divide_cc ()
fg.connect (stream1, (divider, 0))
fg.connect (stream2, (divider, 1))
fg.connect (stream3, (divider, 2))
fg.connect (divider, outputstream)
```

Then the output of the `subtractor` is stream1÷stream2÷stream3.

4.7 Log function

Block: `gr.nlog10_ff`

Usage:

```
gr.nlog10_ff ( float n,
               unsigned vlen )
```

Notes: `gr.nlog10_ff` computes n ×$\log_{10}$(input). Input and output are both float numbers. Ignore the argument `vlen`, the vector length, using its default value 1.

5 Type Conversions

5.1 Complex Conversions

Block:

```
    gr.complex_to_float
    gr.complex_to_real
    gr.complex_to_imag
    gr.complex_to_mag
    gr.complex_to_arg
```

Usage:

```
    gr.complex_to_float( unsigned int vlen )
    gr.complex_to_real( unsigned int vlen )
    gr.complex_to_imag( unsigned int vlen )
    gr.complex_to_mag( unsigned int vlen )
    gr.complex_to_arg( unsigned int vlen )
```

Notes: The argument `vlen` is the vector length, we almost always use the default value 1. So just ignore the argument. These blocks convert a complex signal to separate real & imaginary float streams, just the real part, just the imaginary part, the magnitude of the complex signal and the phase angle of the complex signal. Note that the block `gr.complex_to_float` could have 1 or 2 outputs. If it is connected to only one output, then the output is the real part of the complex signal. Its effect is equivalent to `gr.complex_to_real`.

5.2 Float Conversions

Block:

```
    gr.float_to_complex
    gr.float_to_short
    gr.short_to_float
```

Usage:

```
    gr.float_to_complex ( )
    gr.float_to_short ( )
    gr.short_to_float ( )
```

Notes: These blocks are like `adapters', used to provide the interface between two blocks with different types. Note that the block `gr.float_to_complex` may have 1 or 2 inputs. If there is only one, then the input signal is the real part of the output signal, while the imaginary part is constant 0. If there are two inputs, they serve as the real and imaginary parts of the output signal respectively.

6 Filters

6.1 FIR Designer

Block: `gr.firdes`

Notes: `gr.firdes` has several static public member functions used to design different types of FIR filters. These functions return a vector containing the FIR coefficients. The returned vector is often used as an argument of other FIR filter blocks.

6.1.1 Low Pass Filter

Usage:

```
vector< float > gr.firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76)   [static]
```

Notes: `low_pass` is a static public member function in the class `gr.firdes`. It designs the FIR filter coefficients (`taps`) given the filter specifications. Here the argument `window` is the type of the window used in the FIR filter design. Valid values include

```
    WIN_HAMMING
    WIN_HANN
    WIN_BLACKMAN
    WIN_RECTANGULAR
```

6.1.2 High Pass Filter

Usage:

```
vector< float > gr.firdes::high_pass ( double      gain,
                                       double      sampling_freq,
                                       double      cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.3 Band Pass Filter

Usage:

```
vector< float > gr.firdes::band_pass ( double      gain,
                                       double      sampling_freq,
                                       double      low_cutoff_freq,
                                       double      high_cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.4 Band Reject Filter

Usage:

```
vector< float > gr.firdes::band_reject ( double      gain,
                                         double      sampling_freq,
                                         double      low_cutoff_freq,
                                         double      high_cutoff_freq,
                                         double      transition_width,
                                         win_type    window = WIN_HAMMING,
                                         double      beta = 6.76)   [static]
```

6.1.5 Hilbert Filter

Usage:

```
vector< float > gr.firdes::hilbert ( unsigned int     ntaps,
                                     win_type    windowtype = WIN_RECTANGULAR,
                                     double      beta = 6.76 )   [static]
```

Notes: `gr.firdes::hilbert` designs a Hilbert transform filter. `ntaps` is the number of taps, which must be odd.

6.1.6 Raised Cosine Filter

Usage:

```
vector< float > gr.firdes::root_raised_cosine ( double      gain,
                                                double      sampling_freq,
                                                double      symbol_rate,
                                                double      alpha,
                                                int         ntaps )   [static]
```

Notes: `gr.firdes::root_raised_cosine` designs a root cosine FIR filter. The argument `alpha` is the excess bandwidth factor. `ntaps` is the number of taps.

6.1.7 Gaussian Filter

Usage:

```
vector< float > gr.firdes::gaussian ( double      gain,
                                      double      sampling_freq,
                                      double      symbol_rate,
                                      double      bt,
                                      int         ntaps )   [static]
```

Notes: `gr.firdes::gaussian` designs a Gaussian filter. The argument `ntaps` is the number of taps.

6.2 FIR Decimation Filters

Block:

```
    gr.fir_filter_ccc
    gr.fir_filter_ccf
    gr.fir_filter_fcc
    gr.fir_filter_fff
    gr.fir_filter_fsf
    gr.fir_filter_scc
```

Usage:

```
    gr.fir_filter_ccc (int decimation,
                       const std::vector< gr_complex > & taps )
    gr.fir_filter_ccf (int decimation,
                       const std::vector< float > & taps )
    gr.fir_filter_fcc (int decimation,
                       const std::vector< gr_complex > & taps )
    gr.fir_filter_fff (int decimation,
                       const std::vector< float > & taps )
    gr.fir_filter_fsf (int decimation,
                       const std::vector< float > & taps )
    gr.fir_filter_scc (int decimation,
                       const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the decimation rate of the FIR filter. If it is 1, then it's just a normal 1:1 FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.3 FIR Interpolation Filters

Block:

```
    gr.interp_fir_filter_ccc
    gr.interp_fir_filter_ccf
    gr.interp_fir_filter_fcc
    gr.interp_fir_filter_fff
    gr.interp_fir_filter_fsf
    gr.interp_fir_filter_scc
```

Usage:

```
    gr.interp_fir_filter_ccc (unsigned interpolation,
                              const std::vector< gr_complex > & taps )
    gr.interp_fir_filter_ccf (unsigned interpolation,
                              const std::vector< float > & taps )
    gr.interp_fir_filter_fcc (unsigned interpolation,
                              const std::vector< gr_complex > & taps )
    gr.interp_fir_filter_fff (unsigned interpolation,
                              const std::vector< float > & taps )
    gr.interp_fir_filter_fsf (unsigned interpolation,
                              const std::vector< float > & taps )
    gr.interp_fir_filter_scc (unsigned interpolation,
                              const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the interpolation rate of the FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.4 Digital Down Converter with FIR Decimation Filters

Block:

```
    gr.freq_xlating_fir_filter_ccc
    gr.freq_xlating_fir_filter_ccf
    gr.freq_xlating_fir_filter_fcc
    gr.freq_xlating_fir_filter_fcf
    gr.freq_xlating_fir_filter_scc
    gr.freq_xlating_fir_filter_scf
```

Usage:

```
gr.freq_xlating_fir_filter_ccc [ccf, fcc, fcf, scc, scf]
                              ( int      decimation,
                                const std::
vector< gr_complex [float] > &  taps,
                                double      center_freq,
                                double      sampling_freq )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

These blocks are used as FIR filters combined with frequency translation. Recall that in tutorial 4, in the FPGA there are digital down converters (DDC) translating the signal from IF to baseband. Then the following decimator downsamples the signal and selects a narrower band of the signal by low-pass filtering it. These blocks do the same things except in software. These classes efficiently combine a frequency translation (typically down conversion) with an FIR filter (typically low-pass) and decimation. It is ideally suited for a `channel selection filter' and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

6.5 Hilbert Transform Filter

Block: `gr.hilbert_fc`

Usage:

```
gr.hilbert_fc( unsigned int ntaps )
```

Notes: `gr.hilbert_fc` is a Hilbert transformer. The real output is input appropriately delayed. Imaginary output is Hilbert filtered (90 degree phase shift) version of input. We only need to specify the number of taps `ntaps` when using the block. The Hilbert filter designer `gr.firdes::hilbert` is used implicitly in the implementation of the block `gr.hilbert_fc`.

6.6 Filter-Delay Combination Filter

Block: `gr.filter_delay_fc`

Usage:

```
gr.filter_delay_fc( const std::vector< float > &  taps )
```

Notes: This is a filter-delay combination block. The block takes one or two float stream and outputs a complex stream. If only one float stream is input, the real output is a delayed version of this input and the imaginary output is the filtered output. If two floats are connected to the input, then the real output is the delayed version of the first input, and the imaginary output is the filtered output. The delay in the real path accounts for the group delay introduced by the filter in the imaginary path. The filter taps needs to be calculated using `gr.firdes` before initializing this block.

6.7 IIR Filter

Block: `gr.iir_filter_ffd`

Usage:

```
gr.iir_filter_ffd ( const std::vector< double > &  fftaps,
                    const std::vector< double > &  fbtaps)
```

Notes: The suffix `ffd` means float input, float output and double taps. This IIR filter uses the Direct Form I implementation, where `fftaps` contains the feed-forward taps, and `fbtaps` the feedback ones. `fftaps` and `fbtaps` must have equal numbers of taps. The input and output satisfy a difference equation of the form

$$y[n]-\sum_{k=1}^{N} a_k y[n-k]=\sum_{k=0}^{M} b_k x[n-k]$$

with the corresponding rational system function

$$H(z)=\frac{\sum_{k=0}^{M} b_k z^{-k}}{1-\sum_{k=1}^{N} a_k z^{-k}}$$

Note that some texts define the system function with a `+' in the denominator. If you're using that convention, you'll need to negate the feedback taps.

6.8 Single Pole IIR Filter

Block: `gr.single_pole_iir_filter_ff`

Usage:

```
gr.single_pole_iir_filter_ff ( double alpha,
                               unsigned int    vlen )
```

Notes: This is a single pole IIR filter with float input, float output. The input and output satisfy a difference equation of the form:

$$y[n]-(1-\alpha)y[n-1]=\alpha x[n]$$

with the corresponding rational system function

$$H(z)=\frac{\alpha}{1-(1-\alpha)z^{-1}}$$

Note that some texts define the system function with a + in the denominator. If you're using that convention, you'll need to negate the feedback tap. The argument `vlen` is the vector length. We usually use its default value 1, so just ignore it.

7 FFT

Block:

```
gr.fft_vcc
gr.fft_vfc
```

Usage:

```
gr.fft_vcc ( int      fft_size,
                bool    forward,
                bool    window )
    gr.fft_vfc ( int     fft_size,
                bool    forward,
                bool    window )
```

Notes: These blocks are used to compute the fast Fourier transforms of the input sequence. For `gr.fft_vcc`, it computes forward or reverse FFT, with complex vector in and complex vector out. For `gr.fft_vfc` it computes forward FFT with float vector in and complex vector out.

8 Other useful blocks

8.1 FM Modulation and Demodulation

Block:

```
    gr.frequency_modulator_fc
    gr.quadrature_demod_cf
```

Usage:

```
    gr.frequency_modulator_fc ( double sensitivity )
    gr.quadrature_demod_cf ( float gain )
```

Notes: The `gr.frequency_modulator_fc` block is the FM modulator. The instantaneous frequency of the output complex signal is proportional to the input float signal. We have seen `gr.quadrature_demod_cf` in the FM receiver example. It calculates the instantaneous frequency of the input signal.

8.2 Numerically Controlled Oscillator

Block: `gr.fxpt_nco`

Usage: This block is used to generate sinusoidal waves. We can set or adjust the phase and frequency of the oscillator by several public member functions of the class. The functions include:

```
    void    set_phase (float angle)
    void    adjust_phase (float delta_phase)
    void    set_freq (float angle_rate)
    void    adjust_freq (float delta_angle_rate)
    void    step ()
    void    step (int n)
    float   get_phase () const
    float   get_freq () const
    void    sincos (float *sinx, float *cosx) const
    float   cos () const
    float   sin () const
```

We use `cos()` or `sin()` function to get the sinusoidal samples. They calculate sin and cos values according to the current phase. The `freq` is actually the phase difference between consecutive samples. When we call `step()` method, the current phase will increase by the `freq`.

8.3 Blocks for digital transmission

Block:

```
    gr.bytes_to_syms
    gr.simple_framer
    gr.simple_correlator
```

Usage:

```
    gr.bytes_to_syms ( )
    gr.simple_framer ( int payload_bytesize )
    gr.simple_correlator ( int payload_bytesize )
```

Notes: `gr.bytes_to_syms ( )` converts a byte sequence (for example, a unsigned char stream) to a ± sequence, i.e. the digital binary symbols. `gr.simple_framer` packs a byte stream into packets, with the length of `payload_bytesize`. Then necessary synchronization and command bytes are added at the head. `gr.simple_correlator` is a digital detector, which synchronizes the symbol and frame, finally detects the digital information correctly. The real implementation of these blocks is a little bit complicated. Please study the FSK example `fsk_r[t]x.py` and the source code of these blocks. These blocks are very helpful when we want to design digital transmission schemes.

9 Wrap up

This tutorial reviews some of the most frequently used blocks in GNU Radio. Using these blocks skillfully would be very important and useful. Certainly we only discuss a fraction of the available blocks. There are some other more advanced, less used blocks that we didn't talk about. Also more and more blocks are coming as GNU Radio gets more popular (maybe including your block one day). Our introduction is very simple. If you wish to know all the details about a block, please go to its documentation page and then read its source code directly. The source code is the best place to understand what's going on in a block thoroughly. Studying some of the examples to see how a block a used is also a very good way.

References

[1]
GNU Radio 2.x Documentation http://www.gnu.org/software/gnuradio/doc/index.html

---

Footnotes:

[1]The author is affiliated with Networking Communications and Information Processing (NCIP) Laboratory, Department of

Electrical Engineering, University of Notre Dame. Welcome to send me your comments or inquiries. Email: dshen@nd.edu

---

Tutorial 10: Writing A Signal Processing Block for GNU Radio – Part I

Dawei Shen[*]

June 11, 2005

Abstract

This article explains how to write signal processing blocks for GNU Radio. Concretely, we will talk about how to implement a class derived from `gr_block` in C++, the naming conventions and how to use SWIG to generate the interface between Python and C++. The final 'product' will be a Python module in *gnuradio* package, allowing us to access the block in a simply way.

1 Overview

In this article, we explain how to write signal processing blocks for GNU Radio. This article is actually an expanded version of the on-line documentation: ' **How to Write a Signal Processing Block**', written by *Eric Blossom*. We will cover the same materials and use the same example. However, more comments and details will be added to improve the readability.

In previous tutorials, we have introduced the 'Python - C++' two-tier structure of GNU Radio. We also have met a few blocks in the examples, such as `gr_sig_source_f`, `gr_quadrature_demod_cf`, etc. However, we skipped the details about how these blocks are implemented in C++ and how they are glued to Python. In this tutorial, all these secrets will be revealed.

This article will walk through the construction of several simple signal processing blocks, and explain the techniques and idioms used.

2 The view from 30000 feet

From the Python's point of view, GNU Radio provides a data flow abstraction. The fundamental concepts are signal processing blocks and the connections between them. This abstraction is implemented by the Python `gr.flow_graph` class, as we have discussed in tutorial 6. Each block has a set of input ports and output ports. Each port has an associated data type. The most common port types are float and `gr_complex` (equivalent to `std::complex<float>`).

From the high level point-of-view, infinite streams of data flow through the ports. At the C++ level, streams are dealt with in convenient sized pieces, represented as contiguous arrays of the underlying type.

When we write the block, we need to construct them as shared libraries that may be dynamically loaded into Python using the 'import' mechanism. **SWIG**, the Simplified Wrapper and Interface Generator, is used to generate the glue that allows our code to be used from Python. Writing a new signal processing block involves creating 3 files: The `.h` and `.cc` files that define the new block class

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

and the `.i` file that tells SWIG how to generate the glue that binds the class into Python. The new class must derive from `gr_block` or one of it's subclasses.

3 The base class of all signal processing blocks: gr_block

The C++ class `gr_block` is the base of all signal processing blocks in GNU Radio. The new block class we try to create must derive from `gr_block` or one of it's subclasses. So let's study it first by looking at `gr_block.h`. To find the source code of `gr_block.h`, we can open the documentation for GNU Radio, choose the 'File List' tag and search for `gr_block.h`. You can also find it located at `/src/lib/runtime`. Refer to appendix A for the source code.

Let's take a glance at the member variables defined in `gr_block` first:

```
private:
    std::string             d_name;
    gr_io_signature_sptr    d_input_signature;
    gr_io_signature_sptr    d_output_signature;
    int                     d_output_multiple;
    double                  d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr    d_detail;       // implementation details
    long                    d_unique_id;        // convenient for debugging
```

`d_name` is a string saving the block's name. `d_unique_id` is a long integer, defined as the 'ID' of the block. This is convenient for debugging purpose.

What are `d_input_signature` and `d_output_signature`? What's the data type `gr_io_signature_sptr`? The story becomes complicated so early. Here we have to explain two issues: the class `gr_io_signature`, and the boost smart pointer. So, let's make some preparations first.

3.1 The class for 'IO signature': gr_io_signature

The class `gr_io_signature` is defined in `/src/lib/runtime/gr_io_signature.h`. As indicated by its name, `gr_io_signature` is like an endorsement on the block's input or output flows, telling their basic information. Let's look at part of its definitions in `gr_io_signature.h`:

```
class gr_io_signature {
public:
    ~gr_io_signature ();
    int min_streams ()  const { return d_min_streams; }
    int max_streams ()  const { return d_max_streams; }
    size_t sizeof_stream_item (int index) const { return d_sizeof_stream_item; }
private:
    int         d_min_streams;
    int         d_max_streams;
    size_t      d_sizeof_stream_item;
    gr_io_signature (int min_streams, int max_streams, size_t sizeof_stream_item);
    friend gr_io_signature_sptr gr_make_io_signature (  int min_streams,
                                                        int max_streams,
                                                        size_t sizeof_stream_item);
};
```

For a block's input (output), the class `gr_io_signature` defines the minimal (`d_min_streams`) and maximal (`d_max_streams`) number of streams as the lower and upper bound. The 'size' (number of bytes occupied) of an item in the stream is given by `d_sizeof_stream_item`, a member variable with the type of `size_t`.

When we create a block, we need to indicate two 'signatures' for both input and output flows.

3.2 Smart pointers: Boost

What is the data type `gr_io_signature_sptr`? Let's make an extensive investigation on it. `gr_runtime.h` is included in `gr_block.h`. In `gr_runtime.h`, we see `gr_io_signature_sptr` is a type defined as:

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

Further, `gr_runtime.h` includes `gr_type.h` first. In `gr_type.h`, we include an interesting header file:

```
#include <boost/shared_ptr.hpp>
```

GNU Radio takes the advantage of `Boost` smart pointers.

3.2.1 What is Boost?

`Boost` is a collection of C++ libraries, a pre-required package for the installation of GNU Radio. `Boost` provides powerful extensions to C++ from many aspects, such as algorithm implementation, math/numerics, input/output, iterators, etc. Interested programmers can refer to http://www.boost.org for more information.

3.2.2 What is smart pointer?

GNU Radio borrows one cool feature from `Boost`: the `smart_ptr` library, so called smart pointers. Smart pointers are objects which store pointers to dynamically allocated objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners. Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

In fact, the smart pointers are defined as class templates. The library `smart_ptr` provides five smart pointer class templates, but in GNU Radio, we only use one of them: `shared_ptr`, defined in `<boost/shared_ptr.hpp>`. `shared_ptr` is used for the case when the pointed object ownership is shared by multiple pointers.

3.2.3 How to use the smart pointers in GNU Radio?

To use the smarter pointer `shared_ptr` in GNU Radio, first we need to include the header file `<boost/shared_ptr.hpp>`. Then we can use it to define a smart pointer as:

```
    boost::shared_ptr<T>   pointer_name
```

These smart pointer class templates have a template parameter, T, which specifies the type of the object pointed to by the smart pointer. For example, `boost::shared_ptr<gr_io_signature>` declares a smart pointer pointing to an object with the class type of `gr_io_signature`. Ok, thus far we have explained what `gr_io_signature_sptr` means.

Anyway, we can simply treat the Boost smart pointer as a built-in C++ pointer, though it's equipped with some cool features. There is nothing difficult.

Let's go back to the member variables in the class `gr_block`. Now we can understand `d_input_signature` and `d_output_signature` are two smart pointers referring to `gr_io_signature` objects for the input and output flows, which is the basic information of a block. Let's hang up the other member variables for a while and look at two important methods defined in `gr_block`: `forecast()`, and `general_work()`. Before introducing them, it is better to look at the data types defined in GNU Radio first.

3.3 The data types in GNU Radio

The special data types defined in GNU Radio can be found in `gr_types.h` and `gr_complex.h`:

```
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;
```

As we see, the data types with the prefix '`gr_`' are nothing but new names for the C++ built-in data types. We just represent them using a consistent and convenient way. `vector` and `complex` are both C++ standard libraries, which are quite useful in GNU Radio.

3.4 The core of a block: the method general_work()

The method `general_work()` is pure virtual, we definitely need to override that. `general_work` is the method that does the actual signal processing, which is the 'CPU' of the block.

```
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;
```

Generally speaking, the method `general_work()` compute the output streams from the input streams. Let's introduce the four arguments first.

`noutput_items` is the number of output items to write on each output stream. `ninput_items` gives the number of input items available on each input stream. A block may have `x` input streams and `y` output streams. `ninput_items` is an integer '`vector`' of length `x`, the i^{th} element of which gives the number of available items on the i^{th} input stream. However, for the output flow, why is `noutput_items` simply an integer, not a vector? This is because, due to some technical issues, GNU Radio of the current version only supports a block having the same data rates for all output streams, i.e. the number of output items to write on each output stream is the same for all output streams. The data rates for input streams could be different.

We have introduced the data types `gr_vector_const_void_star` and `gr_vector_void_star` in last subsection. `input_items` is a vector of pointers to the input items, one entry per input stream. `output_items` is a vector of pointers to the output items, one entry per output stream. We actually use these pointers to get the input data and write the computed output data to appropriate streams.

Note that the last three arguments `ninput_items`, `input_items` and `output_items` are passed by reference, indicated by the character '`&`'. So they can possibly be modified in the method `general_work()`.

The returned value of `general_work()` is the number of items actually written to each output stream, or -1 on EOF. It is OK to return a value less than `noutput_items`.

Finally and significantly, when we override `general_work()` for our own block, we **MUST** call `consume()` or `consume_each()` methods to indicate how many items have been consumed on each input stream.

```
void consume (int which_input, int how_many_items);
```

The method `consume()` tells the scheduler how many items (given by '`how_many_items`') of the i^{th} input stream (given by '`which_input`') have been consumed. If each input stream has consumed the same number of items, we can use `consume_each()` instead.

```
void consume_each (int how_many_items);
```

It tells the scheduler each input stream has consumed '`how_many_items`' items.

The reason why we have to call the `consume()` or `consume_each()` method is we have to tell the scheduler how many items of the input streams have been consumed, so that the scheduler can arrange the upstream buffer and associated pointers accordingly. The stories behind these two methods are overdetailed for us. Just keep in mind that they have been well implemented by GNU Radio and we should remember to call them every time we override the `general_work()` method.

3.5 Brief introduction to other methods and member variables

3.5.1 The method forecast()

```
virtual void forecast ( int             noutput_items,
                        gr_vector_int   &ninput_items_required);
```

The method `forecast()` is used to estimate the input requirements given an output request.

The first parameter `noutput_items` has been introduced in `general_work()`, which is the number of output items to produce for each output stream. The second parameter `ninput_items_required` is an integer vector, saving the number of input items required on each input stream.

When we override the `forecast()` method, we need to estimate the number of data items required on each input stream, given the request to produce '`noutput_items`' items for each output stream. The estimate doesn't have to be exact, but should be close. The argument `ninput_items_required` is passed by reference, so that the calculated estimates can be saved into it directly.

We can look at '`/src/lib/runtime/gr_block.cc`' to look at its 'stub 1:1 implementation', in which the number of input items required on each input stream is simply set to `noutput_items`. This is valid for many regular blocks, but obviously not appropriate for interpolators, decimators, or blocks with a more complicated relationship between `noutput_items` and the input requirements.

3.5.2 d_output_multiple and the method set_output_multiple()

Now let's introduce the fourth member variable defined in `gr_block`: `d_output_multiple`. It is used to constrain the `noutput_items` argument passed to `forecast()` and `general_work()`.

The scheduler will ensure that the `noutput_items` argument passed to `forecast()` and `general_work()` will be an integer multiple of `d_output_multiple`. The default value of `d_output_multiple` is 1.

Here is a critical point worth emphasizing. Suppose we're going to design a block class, after we override the `general_work()` or `forecast()` methods, who will use or call these methods, and how? Of particular importance, what value will be passed to the argument `noutput_items` and who does that? A simple answer could be: 'The scheduler will call these methods with appropriate arguments.' When we implement `general_work()` or `forecast()`, we always assume `noutput_items` and other arguments have been provided conceptually. In fact, we never call these methods and set the arguments explicitly. The '`scheduler`' will organize everything and call the methods according to the higher level policy and buffer allocation strategy. The tricks behind the scene involve too many details and are beyond the necessity. Don't worry about them when we design our own blocks:)

We do have some level of control to the argument `noutput_items`. The variable `d_output_multiple` tells the scheduler `noutput_items` must be an integer multiple of `d_output_multiple`.

We can set the value of `d_output_multiple` using the method `set_output_multiple()` and get its value using the method `output_multiple()`.

```
void    gr_block::set_output_multiple (int multiple)
{
    if (multiple < 1)
        throw std::invalid_argument ("gr_block::set_output_multiple");
    d_output_multiple = multiple;
}
int     output_multiple () const { return d_output_multiple; }
```

3.5.3 d_relative_rate and the method set_relative_rate()

The fifth member variable `d_relative_rate` gives the approximate information on the relative data rate, i.e. the approximate output rate / input rate.

This information provides a hint to the buffer allocator and scheduler, so that they can arrange the buffer allocation and adjust parameters accordingly. `d_relative_rate` is 1.0 by default, which is true for most signal processing blocks. Obviously, the decimators' `d_relative_rates` should be less than 1.0, while the interpolators' `d_relative_rates` is larger than 1.0.

We can set the value of `d_relative_rate` using the method `set_relative_rate()` and get its value using the method `relative_rate()`.

```
void    gr_block::set_relative_rate (double relative_rate)
{
    if (relative_rate < 0.0)
        throw std::invalid_argument ("gr_block::set_relative_rate");
    d_relative_rate = relative_rate;
}
double  relative_rate () const { return d_relative_rate; }
```

OK! At this point, we have introduced the class `gr_block` thoroughly. Note that we skip the the introduction to the member variable `d_detail` and its related methods. They are too involved and really for only internal use. We seldom meet them when we design our own block. It is strongly recommended to read the source code carefully to gain a complete understanding of these stuffs.

4 Naming Conventions

After studying the class `gr_block` and reading several source files, we now should summarize the naming conventions used in GNU Radio. Following the name conventions could assist us in comprehending the code base and gluing C++ and Python together.

In GNU Radio, with the exception of macros and other constant values, all identifiers shall be in lower case with '`words_separated_like_this`'. Macros and constant values shall be in `UPPER_CASE`.

4.1 Package prefix

All globally visible names (types, functions, variables, constants, etc.) shall begin with a 'package prefix', followed by an underscore. The bulk of the code in GNU Radio belongs to the '`gr`' package, hence names look like `gr_open_file (...)`.

Large coherent bodies of code may use other package prefixes. Here are some frequently seen ones:

```
gr_:    Almost everything in GNU Radio
usrp_:  Universal Software Radio Peripheral (USRP) related packages
qa_:    Quality Assurance, used for our testing code
```

4.2 Class data members (instance variables)

As we have seen, all class data members shall begin with the prefix '`d_`'.

The big win is when you're staring at a block of code, it's obvious which of the things being assigned to persist outside of the block. This also keeps you from having to be creative with parameter names for methods and constructors. You just use the same name as the member variables, without the '`d_`'.

```
class gr_wonderfulness {
    std::string     d_name;
    double          d_wonderfulness_factor;
public:
    gr_wonderfulness (std::string name, double wonderfulness_factor)
    : d_name (name), d_wonderfulness_factor (wonderfulness_factor)

    ...
};
```

All class static data members shall begin with '`s_`'. We haven't met it so far.

4.3 File names

Each significant class shall be contained in its own files. For example, the declaration of the class `gr_foo` shall be in `gr_foo.h` and the definition in `gr_foo.cc`.

4.4 Suffixes

By convention, we encode the input and output types of signal processing blocks in their name using suffixes. The suffix is typically one or two characters long. Sources and sinks have single character suffixes. Regular blocks that have both inputs and outputs have two character suffixes. The first character indicates the type of the input streams, while the second indicates the type of the output streams. FIR filter blocks have a three character suffix, indicating the type of the inputs, outputs and taps, respectively.

These are the suffix characters and their interpretations:

```
f - single precision floating point
c - complex<float>
s - short (16-bit integer)
i - integer (32-bit integer)
```

In addition, for those cases where the block deals with streams of vectors, we use the character 'v' as the first character of the suffix. An example of this usage is `gr_fft_vcc`. The FFT block takes a vector of complex numbers on its input and produces a vector of complex numbers on its output.

5 Our first block: howto_square_ff

Let's work on our first block `howto_square_ff`. It simply computes the square of the input stream:

$$y[n] = x^2[n] \tag{1}$$

We implement the block class `howto_square_ff` directly derived from `gr_block` and override necessary methods such as `general_work()`. The source code can be found at:

```
/gr-howto-write-a-block/src/lib/howto_square_ff.h (.cc)
```

Please also refer to appendix B and C for the code.

Let's emphasize some of key points in the code.

5.1 The constructor

First let's take a look at its constructor, where some tricks are involved:

In `howto_square_ff.h`:

```
...
typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;
howto_square_ff_sptr howto_make_square_ff ();
class howto_square_ff : public gr_block
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();
    howto_square_ff ();
    ...
}
...
```

In `howto_square_ff.cc`:

```
...
howto_square_ff_sptr howto_make_square_ff ()
{
  return howto_square_ff_sptr (new howto_square_ff ());
}
...
static const int MIN_IN = 1;    // mininum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
          gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
  // nothing else required in this example
}
...
```

We have talked about Boost smart pointers just now. In GNU Radio, we use one of the smart pointers '`boost::shared_ptr`' instead of raw C++ pointers for all access to '`gr_`' blocks (and many other data structures) exclusively. So to avoid accidental use of raw C++ pointers, `howto_square_ff`'s constructor is **private**. Private constructor can't be called outside the class. So the following creation of a `howto_square_ff` instance is illegal:

```
howto_square_ff* test_block = new howto_square_ff()
```

because the constructor of `howto_square_ff` is not public. By doing this, we avoid the possibility that a block object is pointed by a raw C++ pointer.

Instead, we use the `howto_make_square_ff()` function as the public interface for creating new instances. First, it is declared as the friend function of the class `howto_square_ff`, so that it could access all private member variables and methods defined in `howto_square_ff`.

```
friend howto_square_ff_sptr howto_make_square_ff ();
```

Then in the function `howto_make_square_ff()`, we call the private constructor of `howto_square_ff` to create an instance using the `new` command. However, we cast the data type of the returned pointer from the raw C++ pointer to the smart pointer `howto_square_ff_sptr`:

```
howto_square_ff_sptr howto_make_square_ff ()
{
  return howto_square_ff_sptr (new howto_square_ff ());
}
```

By playing these tricks, we actually assure that all '`gr_`' blocks are pointed by the smart pointer `boost::shared_ptr` when being created. The function `howto_make_square_ff()` is used as the public interface for creating new instances.

This trick is played for almost every '`gr_`' block and many other data structures. Let's see the private constructor of `howto_square_ff`:

```
static const int MIN_IN = 1;    // minimum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
          gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float))) {}
```

We recall that the constructor of `gr_block` requires the block's name as the first argument, input and output signatures as the second and third arguments. The 2nd and 3rd arguments are actually smart pointers of the type `gr_io_signature_sptr`, pointing to instances of `gr_io_signature`. Similarly, the constructor of `gr_io_signature` is also private. The friend function `gr_make_io_signature()` is used as the public interface for creating new 'signatures' and it returns the smart pointer `gr_io_signature_sptr` pointing to the signatures. As a convention, the `_sptr` suffix indicates the smart pointer `boost::shared_ptr`.

In our block `howto_square_ff`, we need only 1 input stream and 1 output stream. So the minimum and maximum number of streams for both I/O signatures is simply set to 1. The data type of the items in both input and output flows is 'float'. This is also reflected in the suffix of our block's name: '`_ff`'.

5.2 Overriding the core method: general_work()

```
int howto_square_ff::general_work ( int                         noutput_items,
                                    gr_vector_int               &ninput_items,
                                    gr_vector_const_void_star   &input_items,
                                    gr_vector_void_star         &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
    }

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

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

This is the 'real' signal processing part. Its implementation is quite simple and straightforward. Since our block has only 1 input stream and 1 output stream, the vectors of pointers `input_items` and `output_items` contain only one entry. We compute the items on the output stream one by one:

```
for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
}
```

Don't forget to tell the runtime system how many input items have been consumed on each input stream by calling the `consume()` or `comsume_each()` method. In our 1:1 scenario, the number of items consumed on the input stream is simply equal to the items produced on the output stream: `noutput_items`.

Finally, we return the number of items we have produced. Note that in our example, we don't override the `forecast()` method. Because the default 1:1 implementation is reasonable for our block.

OK! We're done! We have got our first own block `howto_square_ff`!

6 conclusion

Now the remaining task is to establish the connection between C++ and Python. Let's take a break and leave it to the next tutorial.

In this article, we analyzed the basic class for all blocks - `gr_block` detailedly. We also introduced the Boost smart pointers and the naming conventions of GNU Radio along the way. This forms the basis of creating a signal processing block. In the next tutorial, we will talk about the hacking skills to glue the Python and C++ together, so that the block we have created in this article can be used from Python in a simple way.

APPENDIX A: The source code: gr_block.h

```
#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H

#include <gr_runtime.h>
#include <string>

class gr_block {

public:

    virtual ~gr_block ();

    std::string name () const { return d_name; }
    gr_io_signature_sptr input_signature () const  { return d_input_signature; }
    gr_io_signature_sptr output_signature () const { return d_output_signature; }
    long unique_id () const { return d_unique_id; }

    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;

    virtual bool check_topology (int ninputs, int noutputs);

    void set_output_multiple (int multiple);
    int  output_multiple () const { return d_output_multiple; }

    void consume (int which_input, int how_many_items);

    void consume_each (int how_many_items);

    void  set_relative_rate (double relative_rate);
    double relative_rate () const { return d_relative_rate; }

private:

    std::string             d_name;
    gr_io_signature_sptr    d_input_signature;
    gr_io_signature_sptr    d_output_signature;
    int                     d_output_multiple;
    double                  d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr    d_detail;       // implementation details
    long                    d_unique_id;        // convenient for debugging

protected:

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

    void set_input_signature (gr_io_signature_sptr iosig){
            d_input_signature = iosig;
            }

    void set_output_signature (gr_io_signature_sptr iosig){
            d_output_signature = iosig;
            }

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

    long gr_block_ncurrently_allocated ();

#endif /* INCLUDED_GR_BLOCK_H */
```

APPENDIX B: The source code: howto_square_ff.h

```
#ifndef INCLUDED_HOWTO_SQUARE_FF_H
#define INCLUDED_HOWTO_SQUARE_FF_H

#include <gr_block.h>

class howto_square_ff;

typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff : public gr_block
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();
    howto_square_ff ();   // private constructor

public:
    ~howto_square_ff ();  // public destructor
    int general_work (  int                         noutput_items,
                        gr_vector_int               &ninput_items,
                        gr_vector_const_void_star   &input_items,
                        gr_vector_void_star         &output_items);
};

#endif /* INCLUDED_HOWTO_SQUARE_FF_H */
```

APPENDIX C: The source code: howto_square_ff.cc

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

#include <howto_square_ff.h>
#include <gr_io_signature.h>

howto_square_ff_sptr
howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}

static const int MIN_IN = 1;    // mininum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
          gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
    // nothing else required in this example
}

howto_square_ff::~howto_square_ff ()
{
    // nothing else required in this example
}

int  howto_square_ff::general_work (int                         noutput_items,
                                    gr_vector_int               &ninput_items,
                                    gr_vector_const_void_star   &input_items,
                                    gr_vector_void_star         &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];
    for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
    }

    consume_each (noutput_items);

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

References

[1] Eric Blossom, **How to Write a Signal Processing Block**, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2] **Python on-line tutorials**, http://www.python.org/doc/current/tut/

[3] Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

Tutorial 10: Writing A Signal Processing Block for GNU Radio - Part I

Dawei Shen[1]

June 11, 2005

Abstract

This article explains how to write signal processing blocks for GNU Radio. Concretely, we will talk about how to implement a class derived from `gr_block` in C++, the naming conventions and how to use SWIG to generate the interface between Python and C++. The final `product' will be a Python module in *gnuradio* package, allowing us to access the block in a simply way.

Contents

1 Overview

In this article, we explain how to write signal processing blocks for GNU Radio. This article is actually an expanded version of the on-line documentation: `**How to Write a Signal Processing Block**', written by *Eric Blossom*. We will cover the same materials and use the same example. However, more comments and details will be added to improve the readability.

In previous tutorials, we have introduced the `Python - C++' two-tier structure of GNU Radio. We also have met a few blocks in the examples, such as `gr_sig_source_f`, `gr_quadrature_demod_cf`, etc. However, we skipped the details about how these blocks are implemented in C++ and how they are glued to Python. In this tutorial, all these secrets will be revealed.

This article will walk through the construction of several simple signal processing blocks, and explain the techniques and idioms used.

2 The view from 30000 feet

From the Python's point of view, GNU Radio provides a data flow abstraction. The fundamental concepts are signal processing blocks and the connections between them. This abstraction is implemented by the Python `gr.flow_graph` class, as we have discussed in tutorial 6. Each block has a set of input ports and output ports. Each port has an associated data type. The most common port types are float and `gr_complex` (equivalent to `std::complex<float>`).

From the high level point-of-view, infinite streams of data flow through the ports. At the C++ level, streams are dealt with in convenient sized pieces, represented as contiguous arrays of the underlying type.

When we write the block, we need to construct them as shared libraries that may be dynamically loaded into Python using the `import' mechanism. **SWIG**, the Simplified Wrapper and Interface Generator, is used to generate the glue that allows our code to be used from Python. Writing a new signal processing block involves creating 3 files: The `.h` and `.cc` files that define the new block class and the `.i` file that tells SWIG how to generate the glue that binds the class into Python. The new class must derive from `gr_block` or one of it's subclasses.

3 The base class of all signal processing blocks: gr_block

The C++ class `gr_block` is the base of all signal processing blocks in GNU Radio. The new block class we try to create must derive from `gr_block` or one of it's subclasses. So let's study it first by looking at `gr_block.h`. To find the source code of `gr_block.h`, we can open the documentation for GNU Radio, choose the `File List' tag and search for `gr_block.h`. You can also find it located at `/src/lib/runtime`. Refer to appendix A for the source code.

Let's take a glance at the member variables defined in `gr_block` first:

```
private:
    std::string             d_name;
    gr_io_signature_sptr    d_input_signature;
    gr_io_signature_sptr    d_output_signature;
    int                     d_output_multiple;
    double                  d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr    d_detail;       // implementation details
    long                    d_unique_id;        // convenient for debugging
```

`d_name` is a string saving the block's name. `d_unique_id` is a long integer, defined as the `ID' of the block. This is convenient for debugging purpose.

What are `d_input_signature` and `d_output_signature`? What's the data type `gr_io_signature_sptr`? The story becomes complicated so early. Here we have to explain two issues: the class `gr_io_signature`, and the boost smart pointer. So, let's make some preparations first.

3.1 The class for `IO signature': gr_io_signature

The class `gr_io_signature` is defined in `/src/lib/runtime/gr_io_signature.h`. As indicated by its name, `gr_io_signature` is like an endorsement on the block's input or output flows, telling their basic information. Let's look at part of its definitions in `gr_io_signature.h`:

```
class gr_io_signature {
public:
    ~gr_io_signature ();
    int min_streams ()  const { return d_min_streams; }
    int max_streams ()  const { return d_max_streams; }
    size_t sizeof_stream_item (int index) const { return d_sizeof_stream_item; }
private:
    int          d_min_streams;
    int          d_max_streams;
    size_t       d_sizeof_stream_item;
    gr_io_signature (int min_streams, int max_streams, size_t sizeof_stream_item);
    friend gr_io_signature_sptr gr_make_io_signature (  int min_streams,
                                                        int max_streams,
                                                        size_t sizeof_stream_item);
};
```

For a block's input (output), the class `gr_io_signature` defines the minimal (`d_min_streams`) and maximal (`d_max_streams`) number of streams as the lower and upper bound. The `size' (number of bytes occupied) of an item in the stream is given by `d_sizeof_stream_item`, a member variable with the type of `size_t`.

When we create a block, we need to indicate two `signatures' for both input and output flows.

3.2 Smart pointers: Boost

What is the data type `gr_io_signature_sptr`? Let's make an extensive investigation on it. `gr_runtime.h` is included in `gr_block.h`. In `gr_runtime.h`, we see `gr_io_signature_sptr` is a type defined as:

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

Further, `gr_runtime.h` includes `gr_type.h` first. In `gr_type.h`, we include an interesting header file:

```
#include <boost/shared_ptr.hpp>
```

GNU Radio takes the advantage of `Boost` smart pointers.

3.2.1 What is Boost?

`Boost` is a collection of C++ libraries, a pre-required package for the installation of GNU Radio. `Boost` provides powerful extensions to C++ from many aspects, such as algorithm implementation, math/numerics, input/output, iterators, etc. Interested programmers can refer to http://www.boost.org for more information.

3.2.2 What is smart pointer?

GNU Radio borrows one cool feature from `Boost`: the `smart_ptr` library, so called smart pointers. Smart pointers are objects which store pointers to dynamically allocated objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners. Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

In fact, the smart pointers are defined as class templates. The library `smart_ptr` provides five smart pointer class templates, but in GNU Radio, we only use one of them: `shared_ptr`, defined in `<boost/shared_ptr.hpp>`. `shared_ptr` is used for the case when the pointed object ownership is shared by multiple pointers.

3.2.3 How to use the smart pointers in GNU Radio?

To use the smarter pointer `shared_ptr` in GNU Radio, first we need to include the header file `<boost/shared_ptr.hpp>`. Then we can use it to define a smart pointer as:

```
    boost::shared_ptr<T>   pointer_name
```

These smart pointer class templates have a template parameter, T, which specifies the type of the object pointed to by the smart pointer. For example, `boost::shared_ptr<gr_io_signature>` declares a smart pointer pointing to an object with the class type of `gr_io_signature`. Ok, thus far we have explained what `gr_io_signature_sptr` means.

Anyway, we can simply treat the Boost smart pointer as a built-in C++ pointer, though it's equipped with some cool features. There is nothing difficult.

Let's go back to the member variables in the class `gr_block`. Now we can understand `d_input_signature` and `d_output_signature` are two smart pointers referring to `gr_io_signature` objects for the input and output flows, which is the basic information of a block. Let's hang up the other member variables for a while and look at two important methods defined in `gr_block`: `forecast()`, and `general_work()`. Before introducing them, it is better to look at the data types defined in GNU Radio first.

3.3 The data types in GNU Radio

The special data types defined in GNU Radio can be found in `gr_types.h` and `gr_complex.h`:

```
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;
```

As we see, the data types with the prefix `` `gr_' `` are nothing but new names for the C++ built-in data types. We just represent them using a consistent and convenient way. `vector` and `complex` are both C++ standard libraries, which are quite useful in GNU Radio.

3.4 The core of a block: the method general_work()

The method `general_work()` is pure virtual, we definitely need to override that. `general_work` is the method that does the actual signal processing, which is the `CPU' of the block.

```
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;
```

Generally speaking, the method `general_work()` compute the output streams from the input streams. Let's introduce the four arguments first.

`noutput_items` is the number of output items to write on each output stream. `ninput_items` gives the number of input items available on each input stream. A block may have `x` input streams and `y` output streams. `ninput_items` is an integer `` `vector' `` of length `x`, the ith element of which gives the number of available items on the ith input stream. However, for the output flow, why is `noutput_items` simply an integer, not a vector? This is because, due to some technical issues, GNU Radio of the current version only supports a block having the same data rates for all output streams, i.e. the number of output items to write on each output stream is the same for all output streams. The data rates for input streams could be different.

We have introduced the data types `gr_vector_const_void_star` and `gr_vector_void_star` in last subsection. `input_items` is a vector of pointers to the input items, one entry per input stream. `output_items` is a vector of pointers to the output items, one entry per output stream. We actually use these pointers to get the input data and write the computed output data to appropriate streams.

Note that the last three arguments `ninput_items`, `input_items` and `output_items` are passed by reference, indicated by the character `&'. So they can possibly be modified in the method `general_work()`.

The returned value of `general_work()` is the number of items actually written to each output stream, or -1 on EOF. It is OK to return a value less than `noutput_items`.

Finally and significantly, when we override `general_work()` for our own block, we **MUST** call `consume()` or `consume_each()` methods to indicate how many items have been consumed on each input stream.

```
void consume (int which_input, int how_many_items);
```

The method `consume()` tells the scheduler how many items (given by `` `how_many_items' ``) of the ith input stream (given by `` `which_input' ``) have been consumed. If each input stream has consumed the same number of items, we can use `consume_each()` instead.

```
void consume_each (int how_many_items);
```

It tells the scheduler each input stream has consumed `` `how_many_items' `` items.

The reason why we have to call the `consume()` or `consume_each()` method is we have to tell the scheduler how many items of the input streams have been consumed, so that the scheduler can arrange the upstream buffer and associated pointers accordingly. The stories behind these two methods are overdetailed for us. Just keep in mind that they have been well implemented by GNU Radio and we should remember to call them every time we override the `general_work()` method.

3.5 Brief introduction to other methods and member variables

3.5.1 The method forecast()

```
virtual void forecast ( int               noutput_items,
                        gr_vector_int     &ninput_items_required);
```

The method `forecast()` is used to estimate the input requirements given an output request.

The first parameter noutput_items has been introduced in general_work(), which is the number of output items to produce for each output stream. The second parameter ninput_items_required is an integer vector, saving the number of input items required on each input stream.

When we override the forecast() method, we need to estimate the number of data items required on each input stream, given the request to produce `noutput_items' items for each output stream. The estimate doesn't have to be exact, but should be close. The argument ninput_items_required is passed by reference, so that the calculated estimates can be saved into it directly.

We can look at `/src/lib/runtime/gr_block.cc' to look at its `stub 1:1 implementation', in which the number of input items required on each input stream is simply set to noutput_items. This is valid for many regular blocks, but obviously not appropriate for interpolators, decimators, or blocks with a more complicated relationship between noutput_items and the input requirements.

3.5.2 d_output_multiple and the method set_output_multiple()

Now let's introduce the fourth member variable defined in gr_block: d_output_multiple. It is used to constrain the noutput_items argument passed to forecast() and general_work().

The scheduler will ensure that the noutput_items argument passed to forecast() and general_work() will be an integer multiple of d_output_multiple. The default value of d_output_multiple is 1.

Here is a critical point worth emphasizing. Suppose we're going to design a block class, after we override the general_work() or forecast() methods, who will use or call these methods, and how? Of particular importance, what value will be passed to the argument noutput_items and who does that? A simple answer could be: `The scheduler will call these methods with appropriate arguments.' When we implement general_work() or forecast (), we always assume noutput_items and other arguments have been provided conceptually. In fact, we never call these methods and set the arguments explicitly. The `scheduler' will organize everything and call the methods according to the higher level policy and buffer allocation strategy. The tricks behind the scene involve too many details and are beyond the necessity. Don't worry about them when we design our own blocks:)

We do have some level of control to the argument noutput_items. The variable d_output_multiple tells the scheduler noutput_items must be an integer multiple of d_output_multiple.

We can set the value of d_output_multiple using the method set_output_multiple() and get its value using the method output_multiple().

```
void    gr_block::set_output_multiple (int multiple)
{
    if (multiple < 1)
        throw std::invalid_argument ("gr_block::set_output_multiple");
    d_output_multiple = multiple;
}
int     output_multiple () const { return d_output_multiple; }
```

3.5.3 d_relative_rate and the method set_relative_rate()

The fifth member variable d_relative_rate gives the approximate information on the relative data rate, i.e. the approximate output rate / input rate.

This information provides a hint to the buffer allocator and scheduler, so that they can arrange the buffer allocation and adjust parameters accordingly. d_relative_rate is 1.0 by default, which is true for most signal processing blocks. Obviously, the decimators' d_relative_rates should be less than 1.0, while the interpolators' d_relative_rates is larger than 1.0.

We can set the value of d_relative_rate using the method set_relative_rate() and get its value using the method relative_rate().

```
void    gr_block::set_relative_rate (double relative_rate)
{
    if (relative_rate < 0.0)
        throw std::invalid_argument ("gr_block::set_relative_rate");
    d_relative_rate = relative_rate;
}
double  relative_rate () const { return d_relative_rate; }
```

OK! At this point, we have introduced the class `gr_block` thoroughly. Note that we skip the the introduction to the member variable `d_detail` and its related methods. They are too involved and really for only internal use. We seldom meet them when we design our own block. It is strongly recommended to read the source code carefully to gain a complete understanding of these stuffs.

4 Naming Conventions

After studying the class `gr_block` and reading several source files, we now should summarize the naming conventions used in GNU Radio. Following the name conventions could assist us in comprehending the code base and gluing C++ and Python together.

In GNU Radio, with the exception of macros and other constant values, all identifiers shall be in lower case with `` `words_separated_like_this' ``. Macros and constant values shall be in `UPPER_CASE`.

4.1 Package prefix

All globally visible names (types, functions, variables, constants, etc.) shall begin with a `package prefix', followed by an underscore. The bulk of the code in GNU Radio belongs to the `` `gr' `` package, hence names look like `gr_open_file (...)`.

Large coherent bodies of code may use other package prefixes. Here are some frequently seen ones:

```
    gr_:    Almost everything in GNU Radio
    usrp_:  Universal Software Radio Peripheral (USRP) related packages
    qa_:    Quality Assurance, used for our testing code
```

4.2 Class data members (instance variables)

As we have seen, all class data members shall begin with the prefix `` `d_' ``.

The big win is when you're staring at a block of code, it's obvious which of the things being assigned to persist outside of the block. This also keeps you from having to be creative with parameter names for methods and constructors. You just use the same name as the member variables, without the `` `d_' ``.

```
class gr_wonderfulness {
    std::string     d_name;
    double          d_wonderfulness_factor;
public:
    gr_wonderfulness (std::string name, double wonderfulness_factor)
    : d_name (name), d_wonderfulness_factor (wonderfulness_factor)

    ...
};
```

All class static data members shall begin with `s_'. We haven't met it so far.

4.3 File names

Each significant class shall be contained in its own files. For example, the declaration of the class `gr_foo` shall be in `gr_foo.h` and the definition in `gr_foo.cc`.

That's the rule.

hehe.

4.4 Suffixes

By convention, we encode the input and output types of signal processing blocks in their name using suffixes. The suffix is typically one or two characters long. Sources and sinks have single character suffixes. Regular blocks that have both inputs and outputs have two character suffixes. The first character indicates the type of the input streams, while the second indicates the type of the output streams. FIR filter blocks have a three character suffix, indicating the type of the inputs, outputs and taps, respectively.

These are the suffix characters and their interpretations:

```
f - single precision floating point
c - complex<float>
s - short (16-bit integer)
i - integer (32-bit integer)
```

In addition, for those cases where the block deals with streams of vectors, we use the character `v' as the first character of the suffix. An example of this usage is `gr_fft_vcc`. The FFT block takes a vector of complex numbers on its input and produces a vector of complex numbers on its output.

5 Our first block: howto_square_ff

Let's work on our first block `howto_square_ff`. It simply computes the square of the input stream:

$$y[n]=x^2[n] \quad (1)$$

We implement the block class `howto_square_ff` directly derived from `gr_block` and override necessary methods such as `general_work()`. The source code can be found at:

```
/gr-howto-write-a-block/src/lib/howto_square_ff.h (.cc)
```

Please also refer to appendix B and C for the code.

Let's emphasize some of key points in the code.

5.1 The constructor

First let's take a look at its constructor, where some tricks are involved:

In `howto_square_ff.h`:

```
...
typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;
howto_square_ff_sptr howto_make_square_ff ();
class howto_square_ff : public gr_block
```

```
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();
    howto_square_ff ();
    ...
}
...
```

In `howto_square_ff.cc`:

```
...
howto_square_ff_sptr howto_make_square_ff ()
{
  return howto_square_ff_sptr (new howto_square_ff ());
}
...
static const int MIN_IN = 1;    // mininum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
          gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
  // nothing else required in this example
}
...
```

We have talked about Boost smart pointers just now. In GNU Radio, we use one of the smart pointers `` `boost::shared_ptr `` ' instead of raw C++ pointers for all access to `` `gr_ `` ' blocks (and many other data structures) exclusively. So to avoid accidental use of raw C++ pointers, `howto_square_ff`'s constructor is **private**. Private constructor can't be called outside the class. So the following creation of a `howto_square_ff` instance is illegal:

```
howto_square_ff* test_block = new howto_square_ff()
```

because the constructor of `howto_square_ff` is not public. By doing this, we avoid the possibility that a block object is pointed by a raw C++ pointer.

Instead, we use the `howto_make_square_ff()` function as the public interface for creating new instances. First, it is declared as the friend function of the class `howto_square_ff`, so that it could access all private member variables and methods defined in `howto_square_ff`.

```
friend howto_square_ff_sptr howto_make_square_ff ();
```

Then in the function `howto_make_square_ff()`, we call the private constructor of `howto_square_ff` to create an instance using the `new` command. However, we cast the data type of the returned pointer from the raw C++ pointer to the smart pointer `howto_square_ff_sptr`:

```
howto_square_ff_sptr howto_make_square_ff ()
{
        return howto_square_ff_sptr (new howto_square_ff ());
      }
```

By playing these tricks, we actually assure that all `gr_' blocks are pointed by the smart pointer `boost::shared_ptr` when being created. The function `howto_make_square_ff()` is used as the public interface for creating new instances.

This trick is played for almost every `gr_' block and many other data structures. Let's see the private constructor of `howto_square_ff`:

```
    static const int MIN_IN = 1;     // minimum number of input streams
    static const int MAX_IN = 1;     // maximum number of input streams
    static const int MIN_OUT = 1;    // minimum number of output streams
    static const int MAX_OUT = 1;    // maximum number of output streams

    howto_square_ff::howto_square_ff ()
      : gr_block ("square_ff",
              gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
              gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float))) {}
```

We recall that the constructor of `gr_block` requires the block's name as the first argument, input and output signatures as the second and third arguments. The 2nd and 3rd arguments are actually smart pointers of the type `gr_io_signature_sptr`, pointing to instances of `gr_io_signature`. Similarly, the constructor of `gr_io_signature` is also private. The friend function `gr_make_io_signature()` is used as the public interface for creating new `signatures' and it returns the smart pointer `gr_io_signature_sptr` pointing to the signatures. As a convention, the `_sptr` suffix indicates the smart pointer `boost::shared_ptr`.

In our block `howto_square_ff`, we need only 1 input stream and 1 output stream. So the minimum and maximum number of streams for both I/O signatures is simply set to 1. The data type of the items in both input and output flows is `float'. This is also reflected in the suffix of our block's name: `_ff'.

5.2 Overriding the core method: general_work()

```
int howto_square_ff::general_work ( int                         noutput_items,
                                    gr_vector_int               &ninput_items,
                                    gr_vector_const_void_star   &input_items,
                                    gr_vector_void_star         &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
    }

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

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

This is the `real' signal processing part. Its implementation is quite simple and straightforward. Since our block has only 1 input stream and 1 output stream, the vectors of pointers `input_items` and `output_items` contain only one entry. We compute the items on the output stream one by one:

```
    for (int i = 0; i < noutput_items; i++){
            out[i] = in[i] * in[i];
    }
```

Don't forget to tell the runtime system how many input items have been consumed on each input stream by calling the `consume()` or `comsume_each()` method. In our 1:1 scenario, the number of items consumed on the input stream is simply equal to the items produced on the output stream: `noutput_items`.

Finally, we return the number of items we have produced. Note that in our example, we don't override the `forecast()` method. Because the default 1:1 implementation is reasonable for our block.

OK! We're done! We have got our first own block `howto_square_ff`!

6 conclusion

Now the remaining task is to establish the connection between C++ and Python. Let's take a break and leave it to the next tutorial.

In this article, we analyzed the basic class for all blocks - `gr_block` detailedly. We also introduced the Boost smart pointers and the naming conventions of GNU Radio along the way. This forms the basis of creating a signal processing block. In the next tutorial, we will talk about the hacking skills to glue the Python and C++ together, so that the block we have created in this article can be used from Python in a simple way.

APPENDIX A: The source code: gr_block.h

```
#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H

#include <gr_runtime.h>
#include <string>

class gr_block {

public:

    virtual ~gr_block ();

    std::string name () const { return d_name; }
    gr_io_signature_sptr input_signature () const  { return d_input_signature; }
    gr_io_signature_sptr output_signature () const { return d_output_signature; }
    long unique_id () const { return d_unique_id; }

    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;

    virtual bool check_topology (int ninputs, int noutputs);

    void set_output_multiple (int multiple);
    int  output_multiple () const { return d_output_multiple; }

    void consume (int which_input, int how_many_items);

    void consume_each (int how_many_items);

    void  set_relative_rate (double relative_rate);

    double relative_rate () const { return d_relative_rate; }

private:

    std::string             d_name;
    gr_io_signature_sptr    d_input_signature;
    gr_io_signature_sptr    d_output_signature;
    int                     d_output_multiple;
    double                  d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr    d_detail;       // implementation details
    long                    d_unique_id;        // convenient for debugging

protected:

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

    void set_input_signature (gr_io_signature_sptr iosig){
            d_input_signature = iosig;
            }

    void set_output_signature (gr_io_signature_sptr iosig){
            d_output_signature = iosig;
            }

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

    long gr_block_ncurrently_allocated ();

#endif /* INCLUDED_GR_BLOCK_H */
```

APPENDIX B: The source code: howto_square_ff.h

```
#ifndef INCLUDED_HOWTO_SQUARE_FF_H
#define INCLUDED_HOWTO_SQUARE_FF_H

#include <gr_block.h>

class howto_square_ff;

typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff : public gr_block
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();

    howto_square_ff ();   // private constructor

public:
    ~howto_square_ff ();  // public destructor
    int general_work (  int                         noutput_items,
                        gr_vector_int               &ninput_items,
                        gr_vector_const_void_star   &input_items,
                        gr_vector_void_star         &output_items);
};

#endif /* INCLUDED_HOWTO_SQUARE_FF_H */
```

APPENDIX C: The source code: howto_square_ff.cc

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

#include <howto_square_ff.h>
#include <gr_io_signature.h>

howto_square_ff_sptr
howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}

static const int MIN_IN = 1;    // mininum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
  : gr_block ("square_ff",
          gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
         gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
    // nothing else required in this example
}

howto_square_ff::~howto_square_ff ()
{
    // nothing else required in this example
}

int  howto_square_ff::general_work (int                         noutput_items,
                                    gr_vector_int               &ninput_items,
                                    gr_vector_const_void_star   &input_items,
                                    gr_vector_void_star         &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
    }

    consume_each (noutput_items);

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

References

[1]
Eric Blossom, **How to Write a Signal Processing Block**, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2]
Python on-line tutorials, http://www.python.org/doc/current/tut/

[3]
Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

---

Footnotes:

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

---

File translated from T_EX by T_TH, version 3.68.

On 23 Jun 2005, 20:05.

Tutorial 11: Writing A Signal Processing Block for GNU Radio – Part II

Dawei Shen[*]

June 15, 2005

Abstract

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

1 Overview

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

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

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

2 The SWIG file: how.i

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

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

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

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

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

```

%feature("autodoc", "1");       // generate python docstrings

%include "exception.i"
%import "gnuradio.i"            // the common stuff

%{
#include "gnuradio_swig_bug_workaround.h"   // mandatory bug fix
#include "howto_square_ff.h"            // the header file
#include <stdexcept>
%}
// ----------------------------------------------------------------
/*
* GR_SWIG_BLOCK_MAGIC does some behind-the-scenes magic so we can
* access howto_square_ff from Python as howto.square_ff()
* First argument 'howto' is the package prefix.
* Second argument 'square_ff' is the name of the class minus the prefix.
*/
GR_SWIG_BLOCK_MAGIC(howto,square_ff);

/*
 * howto_make_square_ff is the friend function of class howto_square_ff
 * It's the public interface for creating new instances
 */
howto_square_ff_sptr howto_make_square_ff ();

/*class declaration*/
class howto_square_ff : public gr_block
{
private:
  howto_square_ff ();            //private constructor
};
```

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

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

3 Directory layout

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

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

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

Table 1: Directory Layout

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

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

3.1 Necessary autotools for building

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

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

3.1.1 Automake

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

3.1.2 Autoconf

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

3.1.3 Libtool

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

3.2 Build tree vs. Install tree

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

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

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

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

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

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

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

3.3 Make check

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

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

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

4 Modifying configuration and make files

4.1 /src/lib/Makefile.am

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

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

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

```

# Install this stuff so that it ends up as the gnuradio.howto module
# This usually ends up at:
#   ${prefix}/lib/python${python_version}/site-packages/gnuradio

ourpythondir = $(grpythondir)
ourlibdir    = $(grpyexecdir)

INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)

SWIGCPPPYTHONARGS = -noruntime -c++ -python $(PYTHON_CPPFLAGS) \
        -I$(swigincludedir) -I$(grincludedir)

ALL_IFILES =                            \
        $(LOCAL_IFILES)                 \
        $(NON_LOCAL_IFILES)

NON_LOCAL_IFILES =                      \
        $(GNURADIO_CORE_INCLUDEDIR)/swig/gnuradio.i

# The .i file comes here
LOCAL_IFILES =                          \
        howto.i

# These files are built and by SWIG.
# The first is the C++ glue.
# The second is the python wrapper that loads the _howto shared library
# and knows how to call our extensions.

BUILT_SOURCES =                         \
        howto.cc                        \
        howto.py

# This gets howto.py installed in the right place
ourpython_PYTHON =                      \
        howto.py

ourlib_LTLIBRARIES = _howto.la

# These are the source files that go into the shared library
_howto_la_SOURCES =                     \
        howto.cc                        \
        howto_square_ff.cc

# magic flags
_howto_la_LDFLAGS = -module -avoid-version

# link the library against some common swig runtime code and the
# c++ standard library
_howto_la_LIBADD =                      \
        -lgrswigrunpy                   \
        -lstdc++

howto.cc howto.py: howto.i $(ALL_IFILES)
        $(SWIG) $(SWIGCPPPYTHONARGS) -module howto -o howto.cc $<

# These headers get installed in ${prefix}/include/gnuradio
```

```
grinclude_HEADERS =                     \
        howto_square_ff.h

# These swig headers get installed in ${prefix}/include/gnuradio/swig
swiginclude_HEADERS =                   \
        $(LOCAL_IFILES)

MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

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

4.2 topdir/configure.ac

Only the first several lines need to be modified slightly:

```
AC_INIT
AC_PREREQ(2.57)
AC_CONFIG_SRCDIR([src/lib/howto.i])
AM_CONFIG_HEADER(config.h)
AC_CANONICAL_TARGET([])
AM_INIT_AUTOMAKE(gr-howto-write-a-block,0.3)
```

Modify the blue parts only.

4.3 /src/python/Makefile.am

A very simple file.

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

EXTRA_DIST = run_tests.in

TESTS =                 \
     run_tests

noinst_PYTHON =             \
     qa_howto.py
```

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

5 Python testing script: qa_howto.py

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

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

from gnuradio import gr, gr_unittest
import howto

class qa_howto (gr_unittest.TestCase):

    def setUp (self):
        self.fg = gr.flow_graph ()

    def tearDown (self):
        self.fg = None

    def test_001_square_ff (self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = gr.vector_source_f (src_data)
        sqr = howto.square_ff ()
        dst = gr.vector_sink_f ()
        self.fg.connect (src, sqr)
        self.fg.connect (sqr, dst)
        self.fg.run ()
        result_data = dst.data ()
        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.main ()
```

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

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

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

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

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

6 We are done!

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

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

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

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

Excellent! We have a new block working!

7 Conclusion

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

References

[1] Eric Blossom, **How to Write a Signal Processing Block**, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2] **Python on-line tutorials**, http://www.python.org/doc/current/tut/

[3] Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

Tutorial 11: Writing A Signal Processing Block for GNU Radio - Part II

Dawei Shen[1]

June 15, 2005

Abstract

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

Contents

1 Overview

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

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

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

2 The SWIG file: how.i

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

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

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

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

%feature("autodoc", "1");       // generate python docstrings

%include "exception.i"
%import "gnuradio.i"            // the common stuff

%{
#include "gnuradio_swig_bug_workaround.h"   // mandatory bug fix
#include "howto_square_ff.h"            // the header file
#include <stdexcept>
%}

// ----------------------------------------------------------------
/*
* GR_SWIG_BLOCK_MAGIC does some behind-the-scenes magic so we can
* access howto_square_ff from Python as howto.square_ff()
* First argument 'howto' is the package prefix.
* Second argument 'square_ff' is the name of the class minus the prefix.
*/
GR_SWIG_BLOCK_MAGIC(howto,square_ff);

/*
 * howto_make_square_ff is the friend function of class howto_square_ff
 * It's the public interface for creating new instances
 */
howto_square_ff_sptr howto_make_square_ff ();

/*class declaration*/
class howto_square_ff : public gr_block
```

```
{
private:
  howto_square_ff ();             //private constructor
};
```

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

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

3 Directory layout

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

Table 1: Directory Layout

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

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

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

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

3.1 Necessary autotools for building

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

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

3.1.1 Automake

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

3.1.2 Autoconf

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

3.1.3 Libtool

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

3.2 Build tree vs. Install tree

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

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

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

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

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

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

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

3.3 Make check

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

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

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

4 Modifying configuration and make files

4.1 /src/lib/Makefile.am

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

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

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

# Install this stuff so that it ends up as the gnuradio.howto module
# This usually ends up at:
#   ${prefix}/lib/python${python_version}/site-packages/gnuradio

ourpythondir = $(grpythondir)
ourlibdir    = $(grpyexecdir)

INCLUDES = $(STD_DEFINES_AND_INCLUDES) $(PYTHON_CPPFLAGS)

SWIGCPPPYTHONARGS = -noruntime -c++ -python $(PYTHON_CPPFLAGS) \
        -I$(swigincludedir) -I$(grincludedir)

ALL_IFILES =                                \
        $(LOCAL_IFILES)                     \
        $(NON_LOCAL_IFILES)

NON_LOCAL_IFILES =                      \
        $(GNURADIO_CORE_INCLUDEDIR)/swig/gnuradio.i

```

```
# The .i file comes here
LOCAL_IFILES =                          \
        howto.i

# These files are built and by SWIG.
# The first is the C++ glue.
# The second is the python wrapper that loads the _howto shared library
# and knows how to call our extensions.

BUILT_SOURCES =                         \
        howto.cc                        \
        howto.py

# This gets howto.py installed in the right place
ourpython_PYTHON =                      \
        howto.py

ourlib_LTLIBRARIES = _howto.la

# These are the source files that go into the shared library
_howto_la_SOURCES =                     \
        howto.cc                        \
        howto_square_ff.cc

# magic flags
_howto_la_LDFLAGS = -module -avoid-version

# link the library against some common swig runtime code and the
# c++ standard library
_howto_la_LIBADD =                      \
        -lgrswigrunpy                   \
        -lstdc++

howto.cc howto.py: howto.i $(ALL_IFILES)
        $(SWIG) $(SWIGCPPPYTHONARGS) -module howto -o howto.cc $<

# These headers get installed in ${prefix}/include/gnuradio
grinclude_HEADERS =                     \
        howto_square_ff.h

# These swig headers get installed in ${prefix}/include/gnuradio/swig
swiginclude_HEADERS =                   \
        $(LOCAL_IFILES)

```

```
MOSTLYCLEANFILES = $(BUILT_SOURCES) *.pyc
```

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

4.2 topdir/configure.ac

Only the first several lines need to be modified slightly:

```
AC_INIT
AC_PREREQ(2.57)
AC_CONFIG_SRCDIR([src/lib/howto.i])
AM_CONFIG_HEADER(config.h)
AC_CANONICAL_TARGET([])
AM_INIT_AUTOMAKE(gr-howto-write-a-block,0.3)
```

Modify the blue parts only.

4.3 /src/python/Makefile.am

A very simple file.

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

EXTRA_DIST = run_tests.in

TESTS =                 \
     run_tests
noinst_PYTHON =             \
     qa_howto.py
```

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

5 Python testing script: qa_howto.py

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

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

from gnuradio import gr, gr_unittest
import howto

class qa_howto (gr_unittest.TestCase):

    def setUp (self):
        self.fg = gr.flow_graph ()

    def tearDown (self):
        self.fg = None

    def test_001_square_ff (self):
        src_data = (-3, 4, -5.5, 2, 3)
        expected_result = (9, 16, 30.25, 4, 9)
        src = gr.vector_source_f (src_data)
        sqr = howto.square_ff ()
        dst = gr.vector_sink_f ()
        self.fg.connect (src, sqr)
        self.fg.connect (sqr, dst)
        self.fg.run ()
        result_data = dst.data ()
        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)

if __name__ == '__main__':
    gr_unittest.main ()
```

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

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

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

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

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

6 We are done!

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

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

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

Excellent! We have a new block working!

7 Conclusion

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

References

[1]

Eric Blossom, **How to Write a Signal Processing Block**, http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[2]

Python on-line tutorials, http://www.python.org/doc/current/tut/

[3]

Eric Blossom, **Exploring GNU Radio**, http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

---

Footnotes:

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

---

GNU Radio Project

- Homepage of GNU Radio project
- GNU Radio Wiki page
- Discuss-gnuradio Archives
- GNU Radio FAQ
- Suggested Readings
- Universal Software Radio Peripheral on Wiki

GNU Radio People's Homepages

- Eric Blossom - the founder and the overall architect of GNU Radio project
- Matt Ettus - also Ettus Research LLC, who sells USRP
- Chuck Swiger - A lot of examples and a good introduction on module usage
- James Cooley - A guy in MIT media lab, providing many useful codes
- Ilia Mirkin - Another guy in MIT Media lab, providing BPSK implementation

Python Programming

- The on-line tutorial
- Python documentation download page
- All the on-line Python documentation
- A very good Chinese Python introduction book (You have to know Chinese)

wxPython Programming

- The on-line tutorial
- wxPython wiki : Getting Started
- On-line documentation