Source code for gr-blocks: <http://gnuradio.org/redmine/projects/gnuradio/repository/revisions/master/show/gr-blocks/lib>

Guided Tutorials:

<http://gnuradio.org/redmine/projects/gnuradio/wiki/Guided_Tutorials>

Blocks coding style:

<http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide#Public-Header-Files>

GRC (gnuradio-companion) creation:

<http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion#Creating-the-XML-Block-Definition>

Blog on GNURadio internals:

<http://sumitgnuradio.blogspot.com/search/label/How%20Gnuradio%20works>

Some nice tutorials:

<http://files.ettus.com/tutorials/labs/Lab_1-5.pdf>

Create an out of tree module:

Create module: In your selected workspace: ‘gr\_modtool newmod <moduleName>’

Add block: gr\_modtool add –t <type> <blockName>, adhere to block naming conventions in the blocks coding stype (link above) , use –tsource for a src block

(for help on add block: gr\_modtool help add)

edit the .cc .h files to set up ports, forecast, add work code, and qa\* (py) to add test code.

Edit the .xml file (in the grc folder) to add parameter keys, keys for in and out ports. This is for GRC.

(cpp code: <my\_block>\_impl.h, <my\_block>impl.cpp, include/<my\_module>/<my\_block>.h)

Make the module:

* mkdir build
* cd build
* cmake ../ # initially I had to install cmake, also had to install gnuradio-dev (sudo apt-get install gunradio-dev)
* make
* make install #run as sudo, # see notes below on how to locate custom OOT blocks.
* ldconfig (#run as sudo)

Use ctest –V to help debug problems in the test (gives a more verbose output).

OOT module edit cycle:

* edit files
* cd build
* make
* run gnuradio-companion

**Tell GRC how to find custom OOT blocks…Method 2: Configuration File**[**¶**](http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion#Method-2-Configuration-File)

Create or edit **~/.gnuradio/config.conf** and add the following lines:

[grc]

local\_blocks\_path=/path/to/my/blocks

Use the path to the GRC .xml file. Be careful with typing in the path. I ended up actually copying from a terminal window.

Note: This allowed my custom OOT block to appear in GRC but when I tried to use the block the runtime complained (from Python) that it couldn’t import the block. This seems due to the need to have a group of .py and swig files in a common directory named for the block (in my case tutorialqpsk). make install solves this by copying files into such a directory. For example, files are like: \_\_init\_\_.py, \_\_init\_\_.pyc, \_\_init\_\_.pyo, tutorialqpsk\_swig.py, tutorialqpsk\_swig.pyc, tutorialqpsk\_swig.pyo, \_tutorialqpsk\_swig.po. Solve this by using make install.

The local\_blocks\_path can contain multiple paths separated by colons: local\_blocks\_path=/path/to/blocks1:/path/to/blocks2

**To see menus (menu bar) in GRC:**

**Maximize GRC window, then hover over the grc file name at the top.**

**Debugging**

In GRC Help/Parser Errors – shows errors from the GRC .xml file.

gnuRadio debugging tutorial:

* <http://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsDebugging>
* In your python test code, after the relevant imports, print out the process id and wait for a keystroke.
* In another window run gdb and tell it to attach to the python process with the given process id.
* Use ‘sudo gdb –pid=<the printed PID>’ to attach gdb to the test script process.
* At this point you can set breakpoints or whatever in your code. Go back to the python window and hit Enter so it'll continue.
* Alternatively, **use Eclipse to debug** (see example below).

Example:

* edit the python test code (in my case, qa\_my\_qpsk\_demod\_cb.py)
  + add ‘import os’
  + in main function section add:
    - print os.getpid()
    - raw\_input(“Press Enter to continue…”)
  + If you need to then debug the python itself do this:
    - edit the shell script that is used to set up the environment and start the python test and add pdb to start the .py test file (instead of python).
    - Run the script like ‘sh script.sh’ (in my case the shell start up script is qa\_my\_qpsk\_demod\_cb\_test.sh’ in build/python.
* Run cmake (from build directory) to make the debug version of the block:
  + cmake <path\_to\_project\_srcs> -DCMAKE\_BUILD\_TYPE=DEBUG (is this necessary?)
* Run gnome-terminal –e <name of the test startup script> to print the pid and wait. The test startup script is a .sh in build/python. In my case, qa\_my\_qpsk\_demod\_cb\_test.sh.
* Start up eclipse as root. ‘sudo <path to eclipse>’ (see desktop shortcut). Needed to run as root for the attach to process ID (further below) to succeed.
* In eclipse select Run/Debug Configurations/C/C++ Attach to Application. Double-click New configuration and select the PID that was printed in the gnome terminal.
* Open C++ source file for the block and set breakpoint(s).
* In gnome terminal press <Enter> to continue the python test script.
* In Eclipse select Run/Resume. The breakpoints will be hit. ☺ *Note: be sure that the test code is actually calling/exectuing the block you are intending to debug.*
* *Alternatively you can create a flow graph in python and debug it like above. Like this:*
  + *Build a flow graph in python that uses the block you want to debug. (an easy way is to generate from GRC).*
  + *Then add the python code to print pid and wait (as above) to the flow graph.*
  + *Run the flow graph… python <flow graph filename> (eg, my\_fg.py).*
  + *Do the steps as above to attach eclipse to the python process (as in the printed pid), continue the flow graph and resume eclipse. Breakpoints will be hit.*

SWIG tutorial with some debugging info:

* <http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf>
* Run debugger on the Python executable.

**Notes on blocks:**

**GNURadio in-tree block documentation:**

* <http://gnuradio.org/doc/doxygen/index.html>

**Block with no inputs and no outputs:**

* Can define this by setting min and max number of input and output ports to 0.
* Constructor is called by top\_block. But work function is not called. I suppose since there in no work to do.

**Specific to Tek RSA Source block:**

* Add /opt/include to CMakeLists.txt (the one in the lib folder for the Tek RSA source) so that we can find the RSA API header files. This is done by appending to the include\_directories command in CMakeLists.txt.
* All the required RSA API libraries (see the RSA API installation notes) need to be added to the link command. This is done by adding them to the CMakeLists.txt (in the lib folder). I did it like this (there may be better/easier ways):

find\_library(<var> <library name>) # the ‘lib’ prepend in filename is removed…looks for .a or .so

list(APPEND RSA\_API\_Libs ${<var>}) # append the library (full path) to the list collecting them all

unset(<var> CACHE) # clear the cache so next find\_library command will work

(example:

find\_library(ll ippch)

list(APPEND RSA\_API\_Libs ${ll})

unset(ll CACHE)

After finding all necessary libraries and appending them to the list add the whole list to the target\_link\_libraries command by appending the list. Like this for example:

target\_link\_libraries(gnuradio-Tek\_RSA … ${RSA\_API\_Libs})

* In the June 2016 RSA API distribution (Ubuntu 15) we get this build error: /opt/lib/libBaseDSPL.a(DSPLTimer.o): relocation R\_X86\_64\_32 against `\_ZTVN4DSPL9DSPLTimerE' can not be used when making a shared object; recompile with –fPIC

To fix: rebuilt libBaseDSPL.a with –fPIC switch and copied into /opt/lib.

* Set the runtime library path so that GNURadio (via gnuradio-companion or python or however you start it) can find the RSA API libraries. Like this:

      export LD\_LIBRARY\_PATH=/opt/lib

Set in shell rc (~/.bashrc), or set permanently in /etc/environment.

* For IQ Streaming (and IF streaming too for that matter):
  + Source block Parameters/Advanced tab provides user control of Output buffer size. Adjust min output buffer to increase the size of the block that the work() function asks for, can help balance the buffers to prevent overflow.
  + When I stream to an FFT (frequency sink) I’m not certain yet why the sink block asks for a large size block (larger than required for the FFT length. When I adjust the source block min buffer size to prevent overflow I get a lot of latency in the processing (turn off the input signal and wait a while until the displayed signal goes away.

**Building GNURadio from sources:**

1. Get dependent packages, etc. See <http://gnuradio.org/doc/doxygen/build_guide.html>
2. Get the source like this:
   1. git clone --recursive <https://github.com/gnuradio/gnuradio.git>
3. Navigate to the source gnuradio folder and build it all like this:
   1. mkdir build
   2. cd build
   3. cmake ../
   4. make
   5. sudo make install
4. Set PYTHONPATH (on mine it’s /usr/local/lib/python2.7/site-packages
5. Set LD\_LIBRARY\_PATH (on mine it’s /opt/lib:/usr/local/lib)
6. I had a problem with the audio sink not working (Runtime Error: audio\_oss\_sink) apparently due to missing audio support package libasound2-dev. To fix I installed libasound2-dev (sudo apt-get install libasound2-dev). Reran cmake and rebuild GNURadio (make).