

# 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 Amplifier (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>