GNU Radio
An introduction

Jesper M. Kristensen
Department of Electronic Systems
jmk@es.aau.dk

Programmable digital devices
Tuesday 6/3 2007
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "‘Hello World'"
   - FM radio
   - Software development

4. References
Outline

1 Introduction
   • What is GNU Radio
     • Software Radio

2 GNU Radio Architecture
   • Hardware Architecture
   • Software Architecture

3 Programming the GNU Radio
   • GNU Radio "Hello World"
   • FM radio
   • Software development

4 References
A Framework

- An open source software toolkit
- Supports, Linux, Mac OS and Windows
  - Creating signal processing applications
  - Defining waveforms in software
  - Processing waveforms in software
- A hardware platform
  - USRP, universal software radio peripheral, low cost HW platform for preprocessing
    - ADC & DAC
    - FPGA
    - USB 2.0 Interface to Host PC

A framework for building software radio transceivers
Prerequisites

- Using the GNU radio is cross disciplinary
- Requiring know-how in the fields of
  - Computer programming
  - Communications systems
  - Digital signal processing
  - Analog as well as digital hardware
    - Hardware is also open source
    - Schematics are available
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "Hello World"
   - FM radio
   - Software development

4. References
Software Radio

- An implementation technology
- A technique for moving digital signal processing as close as possible to the antenna
  - Replacing rigid Hardware...
  - with flexible software based solutions

Software Radio definition

A software (defined) radio is a radio that includes a transmitter in which the operating parameters of the transmitter, including the frequency range, modulation type or maximum radiated or conducted output power can be altered by making a change in software without making any hardware changes.
Quoting the SDR forum:

Software Defined Radio (SDR) is a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band, multi-functional wireless devices that can be enhanced using software upgrades. As such, SDR can really be considered an enabling technology that is applicable across a wide range of areas within the wireless industry. SDR-enabled devices... can be dynamically programmed in software to reconfigure the characteristics of equipment. In other words, the same piece of "hardware" can be modified to perform different functions at different times...
The SDR Forum has defined the following tiers, describing evolving capabilities in terms of flexibility

- **Tier 0**
  - **The Hardware Radio**: The radio is implemented using hardware components only and cannot be modified except through physical intervention.

- **Tier 1**
  - **Software Controlled Radio (SCR)**: Only the control functions of an SCR are implemented in software - thus only limited functions are changeable using software. Typically this extends to inter-connects, power levels etc. but not to frequency bands and/or modulation types etc.
Defining software radio using tiers...

- **Tier 2**
  
  **Software Defined Radio (SDR):** SDRs provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over a broad frequency range. The frequency bands covered may still be constrained at the front-end requiring a switch in the antenna system.

- **Tier 3**
  
  **Ideal Software Radio (ISR):** ISRs provide dramatic improvement over an SDR by eliminating the analog amplification or heterodyne mixing prior to digital-analog conversion. Programmability extends to the entire system with analog conversion only at the antenna, speaker and microphones.
Defining software radio using tiers...

Tier 4

**Ultimate Software Radio (USR):** USRs are defined for comparison purposes only. It accepts fully programmable traffic and control information and supports a broad range of frequencies, air-interfaces & applications software. It can switch from one air interface format to another in milliseconds, use GPS to track the users location, store money using smartcard technology, or provide video so that the user can watch a local broadcast station or receive a satellite transmission.

**Cognitive radio (CR)** is a form of wireless communication in which a transceiver can intelligently detect which communication channels are in use and which are not, and instantly move into vacant channels while avoiding occupied ones. This optimizes the use of available radio-frequency (RF) spectrum while minimizing interference to other users.
S(D)R and Transceiver architectures

- Two basic transceiver architectures
  - The heterodyne
  - The zero IF

![Diagram of S(D)R and Transceiver architectures]
Two basic transceiver architectures

- The heterodyne
- The zero IF
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "Hello World"
   - FM radio
   - Software development

4. References
GNU Radio block schematic in more detail
The Universal Software Radio Peripheral

### Basic USRP facts

- **4*ADC, 12 bit @ 64MSPS**
- **4*DAC, 14 bit @ 128MSPS**
- **Altera EP1C12 FPGA for preprocessing tasks**
- **Analog and Digital I/O for auxiliary I/O**
- **Supporting**
  - Two Receive daughter boards
  - Two Transmit daughter boards
- **USB 2.0 interface to host PC**
More photos of the USRP
Some more ADC specs

- Sampling signals up 32 MHz (Nyquist sampling)
- IF sampling up to 100MHz
  - If internal buffers bypassed, up to 250MHz
- Full range of ADC is 2Vpp (optimized SNR)
The FPGA

RX/TX paths implemented in the FPGA
- Connections controlled via software
- RX path
- TX path
The FPGA

RX/TX paths implemented in the FPGA
- Connections controlled via software
- RX path
- TX path

Connections controlled via software diagram:
- I (from ADC)
- Q (from ADC)
- Sampling rate fs
- Complex multiplication
- Sinewave
- Co-sinewave
- DECIMATOR (CIC)
- DECIMATOR (CIC)
- 16 bit data to host PC via USB
- NCO (Cordic)
- I (decimated)
- Q (decimated)
The FPGA

RX/TX paths implemented in the FPGA

- Connections controlled via software
- RX path
- TX path

Connections controlled via software
The USB interface

- The USB interface supports (ideally) 32Mbyte/s
- this limits the achievable datarates

Examples of datarates

- Complex samples are interleaved across the USB interface
- Having one channel with complex samples gives 8Mbit/s
- Having multiple channels (up to 4) reduces the rate per channel accordingly
- Finally adding TX channels again reduces achievable rates as combined rates is limited by 32Mbyte/s.
The USB interface

- The USB interface supports (ideally) 32Mbyte/s
- this limits the achievable data rates

Examples of data rates

- Complex samples are interleaved across the USB interface
- Having one channel with complex samples gives 8Mbit/s
- Having multiple channels (up to 4) reduces the rate per channel accordingly
- Finally adding TX channels again reduces achievable rates as combined rates is limited by 32Mbyte/s.

<table>
<thead>
<tr>
<th>I0</th>
<th>Q0</th>
<th>I1</th>
<th>Q1</th>
<th>I2</th>
<th>Q2</th>
<th>I3</th>
<th>Q3</th>
<th>I0</th>
<th>Q0</th>
</tr>
</thead>
</table>

Programmerbare digitale enheder
GNU Radio
The USB interface

- The USB interface supports (ideally) 32Mbyte/s
- this limits the achievable datarates

Examples of datarates

- Complex samples are interleaved across the USB interface
- Having one channel with complex samples gives 8Mbit/s
- Having multiple channels (up to 4) reduces the rate per channel accordingly
- Finally adding TX channels again reduces achievable rates as combined rates is limited by 32Mbyte/s.
Hardware architecture in conclusion

- GNU radio supports up to 4 independent real sampling channels
- or 2 independent complex sampling channels
- 2 independent TX channels are supported
- Achievable rates are limited by the USB 32Mbyte/s.
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "Hello World"
   - FM radio
   - Software development

4. References
A 3 tier architecture

- Python scripting language used for creating "signal flow graphs"
- C++ used for creating signal processing blocks
  - An already existing library of signalling blocks
  - OFDM functionality is currently being added, tested and will be added to the library.
- The scheduler is using Python’s built-in module threading, to control the ‘starting’, ‘stopping’ or ‘waiting’ operations of the signal flow graph.
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "Hello World"
   - FM radio
   - Software development

4. References
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 ()
Hello world step-by-step

#!/usr/bin/env python

Makes the python code executable by executing $ chmod +x "'filename.py'"
Hello world step-by-step

#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

importing nesseary modules modules from GNU radio library
Hello world step-by-step

#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio

def build_graph ():

setting up signal flow graph
Hello world step-by-step

#!/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()
Hello world step-by-step

#!/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()
Hello world step-by-step

#!/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 ()

Setting up sinewaves at 350 and 440 Hz
Hello world step-by-step

```python
#!/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()
```

defining destination to be soundcard
#!/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 ()
Hello world step-by-step

```python
#!/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 ()
```

returning created flow graph
Hello world step-by-step

```python
#!/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 ()
```

Create code to execute the flowgraph
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "‘Hello World'"
   - FM radio
   - Software development

4. References
Listening to FM radio on the GNU radio involves:
- Setting up the USRP
  - Selecting RX channel
  - Setting up decimation factor
  - Setting up possible gain factor
- Defining channel selection filter
- Implementing demodulator
- Defining audio filter
- Sending demodulated output to soundcard.
FM receiver block schematic

- From DDC
  - Channel Filter
- Quadrature demodulator:
  - Delay
  - Conj.
- arctan
- Audio filter (De-emphasis)
- To soundcard
Outline

1. Introduction
   - What is GNU Radio
   - Software Radio

2. GNU Radio Architecture
   - Hardware Architecture
   - Software Architecture

3. Programming the GNU Radio
   - GNU Radio "Hello World"
   - FM radio
   - Software development

4. References
Software development on GNU Radio

- The application of Python
  - Using Python for creating flow graphs
  - Also used for creating GUI’s
  - and other non performance critical applications
- The application of C++
  - Performance critical applications
    - Signal processing blocks
    - like the FM demodulator
Signal processing blocks in C++

- Built as shared libraries
- Dynamically loaded using python 'import' feature
- SWIG "'Simplified Wrapper and Interface Generator'" used for glue code allowing python import.
- C++ class "'gr_block'" is base for all signal processing blocks
  - Derived classes gr_sync_block, gr_sync_decimator and gr_interpolator can also be used depending on input/output rate relations
Components needed in writing a C++ block

- .h file for class declaration
- .cc file for class definition
- .i file defining how SWIG generate glue code binding the C++ class into Python
Building a test framework

- Using `gr_unittest` an extension to python `unittest`
- enabling checking for approximate equality tuples of float and complex type
- enabling automated test run’s

assuming block to be verified is ""howto_square_ff"
assuming input and output specifications known
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 ()
For Further Reading

- GNU Radio website
  http://www.gnu.org/software/gnuradio/

- GNU Radio tutorials
  http://www.nd.edu/~jnl/sdr/

- How to Write a Signal Processing Block

- Exploring GNU Radio