

Audio Streaming over FM band between USRP1 and USRP2

Phase II Project Report

Kushal Shah

Department of Electrical and Computer Engineering

EEC 687

Cleveland State University
Cleveland, Ohio 44115

May 7, 2009

1. Abstract

Software Defined Radio is expected to become the dominant technology in radio communications as it provides flexibility in implementation of various radio communication systems by transferring all signal processing runtime and processing blocks to software. This approach is also cost-effective as it only requires low cost external RF hardware and general purpose processors. Much has been done in the field of software radio using GNU Radio as a free open source software development toolkit and USRP1 as a flexible hardware platform to implement digital baseband and IF sections of a radio communication system [1]. Based on the huge success of USRP1, USRP2 was launched recently with more advanced features than USRP1. However, very little research work has been carried out in the field of software radio using USRP2. This motivated me to implement wireless communication between USRP2 and USRP1. In this project, an audio file is streamed over FM band from one host machine to another over a wireless channel using GNU Radio and USRPs. This can be later used as a base to realize data communication at Wi-Fi frequencies between USRP1 and USRP2.

2. Introduction

Software Defined Radio is gaining more importance because it allows a radio to be produced that can receive and transmit different forms of radio protocols just by running different software [2]. The two key components that are being used today to implement software radio are GNU Radio and USRP. The main goal of Software Defined Radio is to make complicated radio functions such as modulation, demodulation, etc., of the signal hardware independent. Thus, the radio system then only requires low cost hardware to receive and transmit the signal. Moreover, the software radio allows building decentralized communication systems [3].

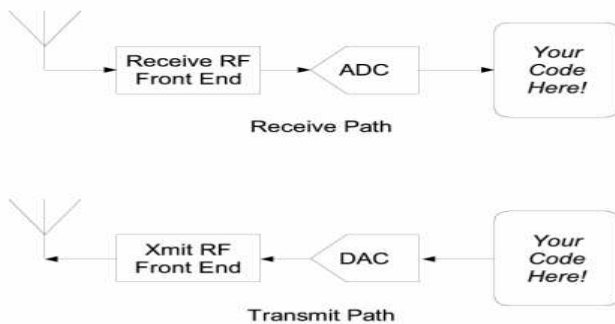


Figure 1 - Typical software radio block diagram [3]

3. Background

3.1. GNU Radio

GNU Radio provides a library of signal processing blocks and the glue to tie it all together [3]. 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 [3]. The signal processing blocks are implemented in C++ [3]. Graphs are constructed and run in Python [3].

GNU Radio uses SWIG (Simplified Wrapper and Interface Generator) to generate the interface between Python and C++.

3.2. USRPs

USRP1 and USRP2 provide the hardware platform for SDR in order to receive and transmit the signal. The USRP motherboard consists of an FPGA, an ADC/DAC and a USB controller. The USRP motherboard can support a variety of daughterboards to achieve wireless communication at different frequencies. It is built on the success of USRP1 but it is not meant to replace USRP1. The following new features are added to USRP2:

- Gigabit Ethernet interface
- 25 MHz of instantaneous RF bandwidth
- Xilinx Spartan 3-2000 FPGA
- Dual 100 MHz 14-bit ADCs
- Dual 400 MHz 16-bit DACs
- 1 MByte of high-speed SRAM
- Locking to an external 10 MHz reference
- 1 PPS (pulse per second) input
- Configuration stored on standard SD cards
- Standalone operation
- The ability to lock multiple systems together for MIMO
- Compatibility with all the same daughterboards as the original USRP [4]

The following table provides comparison between USRP1 and USRP2:

	USRP1	USRP2
Interface	USB 2.0	Gigabit Ethernet
FPGA	Altera EP1C12	Xilinx Spartan 3 2000
RF Bandwidth to/from host	8 MHz @ 16bits	25 MHz @ 16bits
Cost	\$700	\$1400
ADC Samples	12-bit, 64 MS/s	14-bit, 100 MS/s
DAC Samples	14-bit, 128 MS/s	16-bit, 400 MS/s
Daughterboard capacity	2 TX, 2 RX	1 TX, 1 RX
SRAM	None	1 Megabyte
Power	6V, 3A	6V, 3A

Table 1 - Comparison between USRP1 and USRP2 [4]

3.3. Operating System

For this project, Ubuntu 8.10 is being used as the operating system to work with GNU Radio. The reason behind selecting a Debian based Linux operating system is because it is very user friendly and robust.

4. Previous Work

Many wireless applications have been developed using USRP1 and GNU Radio. There are also other applications which only require GNU Radio. However, very few applications have been

developed using USRP2. There are some applications that have already been developed related to this project.

An FM radio transmitter has been realized using GNU Radio and USRP1. In this application, an audio file is converted into a raw file using SOX and frequency modulated using GNU Radio [5]. Finally, it is broadcasted over FM band. The FM receiver is realized for both USRP1 and USRP2 and it is included in the svn version of GNU Radio package [6].

These applications are used as the basis for this project. The transmitter application was developed specifically for USRP1 so it required some modification for use on USRP2.

5. Project

5.1. Project Goal

The aim of this project is to stream an audio file over FM band from one host machine to another using GNU Radio and USRPs. The main purpose of this project is to realize wireless communication involving USRP2 as very little development has been done using USRP2. This would then provide a platform for exploring several other applications for USRP2.

5.2. Project Requirements

The project requirements are listed below:

1. Ubuntu 8.10 – It is a Debian based Linux operating system. There are no strict requirements for the version of Ubuntu in this project. However, the recent version is preferable. The instructions for installation of Ubuntu are provided in [7].
2. GNU Radio (svn) – The latest svn version of GNU Radio is required for this project because only the svn version provides support for USRP2. The current stable release, GNU Radio 3.1.3, does not support USRP2. The installation instructions for svn version of GNU Radio are provided in [1].
3. USRPs – USRP1 and USRP2 provide the radio front-end for transmission and reception of the audio file. The datasheets of USRP1 and USRP2 are available from [8].
4. Daughterboards and antennas – Basic TX, Basic RX and TVRX daughterboards are required for this project. Compatible antennas and a loop back cable are also required. The specifications of daughterboards and antennas are available from [8].
5. SOX – Sound Exchange application is required to convert the audio file into a raw file to enable transmission and reception over USRPs. However, note that SOX must be installed such that it supports the desired audio file format. The installation instructions for SOX can be obtained from [9].

5.3. Project Plan

The project is divided into two phases to ensure that the goal of the project is satisfactorily achieved. In phase I, audio streaming is achieved using USRP1 as the transmitter with USRP2 as the receiver. Phase I is again divided into two parts. In the first part, a loop back cable is used for audio streaming. In the later part, audio streaming is done over a wireless channel. Phase II is similar to phase I except that USRP2 is used as the transmitter while USRP1 as the receiver. Phase I is easy to realize as it has been implemented as described in previous

sections. However, phase II is difficult as it requires Python code development for USRP2. The aim of phase I is to duplicate the previous work and ensure functionality. The experience gained in phase I is used to realize phase II.

5.4. Project Implementation

In this project, a wav file is used for audio streaming. However, there are no restrictions on the audio file format as long as SOX is installed with support for that particular audio file format. Wav file is used in this project because the default installation of SOX supports wav format. It is needed to create a playlist of the audio files. The playlist should follow .pls format. This is required because the Python script used for transmission of audio file accepts playlist in .pls format only. The Python script used for transmission of audio file is fmradio.py (listed in Appendix A from [5]). In this script, the audio files are extracted from the playlist and are converted into raw format using SOX. These raw files are frequency modulated and passed on to the USRP for transmission. For reception, usrp(2)_wfm_rcv.py (listed in Appendix B from [6]) is used. This Python script is already available in the GNU Radio package. The script demodulates the raw file and passes it to the output audio device. The fmradio.py script is written for USRP1. Thus, it was necessary to modify it to be able to use it for USRP2. The modified script is named as fmradio_usrp2.py (listed in Appendix A). The Python script for reception for USRP1 and USRP2 are already implemented in GNU Radio package.

5.5. Project Setup

This project requires two host machines and two USRPs (USRP1 and USRP2). In the case of audio streaming over the loop back cable, Basic TX and Basic RX daughterboards are used. However, for wireless audio streaming, Basic TX and TVRX daughterboards along with the compatible antennas are used. The experimental set up can be observed in the following figures:

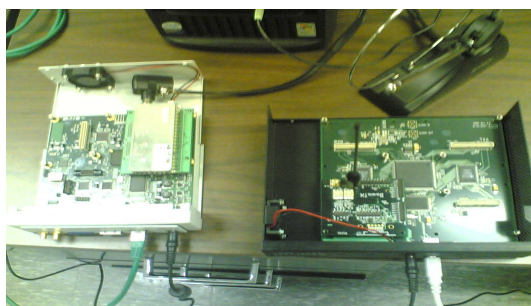


Figure 2 - Audio streaming over wireless channel from USRP1 to USRP2

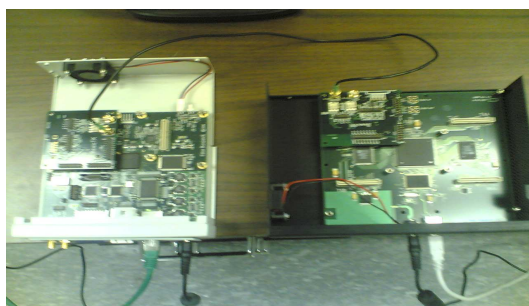


Figure 3 - Audio streaming using loop back cable from USRP2 to USRP1



Figure 4 - Audio streaming over wireless channel from USRP2 to USRP1

Note that the setup for audio streaming using the loop back cable from USRP1 to USRP2 is not shown in the figure.

5.6. Results

Noiseless audio streaming is observed when transmitting from USRP1 to USRP2 over the wireless channel. However, little noise is observed when transmitting from USRP2 to USRP1 over the wireless channel. Performance can be improved by using the loop back cable instead of the wireless channel.

6. Conclusion

This application can be refined more to achieve noiseless audio streaming. It was observed while transmitting from USRP2 that there was some residual frequency which might be inducing noise in the original signal. Moreover, this application can also be upgraded to include real time audio streaming. This project is useful for the beginners who want to understand the basics of USRP and GNU Radio. Experience gained in this project can also be used to achieve data communication over a wireless channel using USRP2.

References

- [1] GNU Radio website, www.gnuradio.org/trac, 2009.
- [2] Software Defined Radio on Wikipedia, www.en.wikipedia.org/wiki/Software-defined_radio, 2009.
- [3] GNU Radio document, www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html, 2004.
- [4] USRP2 on GNU Radio website, www.gnuradio.org/trac/wiki/USRP2, 2009.
- [5] Juha Vierinen, FM Radio transmitter python script, www.mep.fi/viewcvs/fmradio/?root=cvcs, 2009.
- [6] GNU Radio source trunk, www.gnuradio.org/trac/browser/gnuradio/trunk/gnuradio-examples/python, 2009.
- [7] Ubuntu website, www.ubuntu.com, 2009.
- [8] USRP manufacturer's website, www.ettus.com, 2009.
- [9] SOX website, www.sox.sourceforge.net, 2009.

Appendix A

Transmitter python script for USRP1:

fmradio.py [5]:

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

Simple transmit mp3 playlist as a wide band FM signal.

(c) 2007,2009 Juha Vierinen (a lot borrowed from fm_tx4.py in gnuradio-examples)

- Made some modifications so that this runs with new gnuradio (12.03.2009)
- Requires sox in the path
- Only tested on linux
- Probably requires cygwin on windows (nice, killall)

TODO: Create a new chained file source to enable transmission of multiple mp3 playlists simultaneously. This might not be that critical. Still, it would be cool to transmit different mp3 playlists, the 8MHz can fit a lot of WFM channels.

The inverse of this script, a TiVo for FM radio, would be cool too.

```
"""
from gnuradio import gr, eng_notation
from gnuradio import usrp
from gnuradio import audio
from gnuradio import blks2
from gnuradio.eng_option import eng_option
from optparse import OptionParser
from usrpm import usrp_dbid

import math, re, sys, thread, time, tempfile, os, random

def mp3toraw(filename,outputfile):
    print("nice -n 19 sox \"%s\" -r 32000 -t raw -f -L -c 1 %s\n" %
(filename,outputfile))
    os.system("nice -n 19 sox \"%s\" -r 32000 -t raw -f -L -c 1 %s" %
(filename,outputfile))

# Read in .pls format (can be made e.g., using beep-media-player)
def read_playlist(fname):
    input = open(fname, 'r')

    playlist=[]

    #[playlist]
    l = input.readline()
    # NumberOfEntries
    l = input.readline()
    nentries = int(re.findall("NumberOfEntries=([0-9]+)",l)[0])
```

```

print "Number of items in list %d\n" % nentries
i = 1
while 1:
    l=input.readline()

    filepath = re.findall("File[0-9]+=(.*)$",l)
    if filepath:
        print filepath[0]
        playlist.append(filepath[0])
        i = i + 1

input.close()
return(playlist)

## just create a standard tempfn (sox will create the file, so remove one
made by system)
def mktempfn():
    tf = tempfile.mkstemp(".raw")
    outputfile = tf[1]
    os.close(tf[0])
    os.remove(tf[1])
    return(outputfile)

class wfm_tx:
    def __init__(self):

        parser = OptionParser (option_class=eng_option)
        parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
                        help="select USRP Tx side A or B")
        parser.add_option("-f", "--freq", type="eng_float", default=90e6,
                        help="set Tx frequency to FREQ (default 90e6)",
metavar="FREQ")
        parser.add_option("-l","--playlist", action="store", default=None,
                        help="MP3 playlist containing files to air.")
        parser.add_option("-r","--randomize", action="store_true",
default=False,
                        help="Randomize playlist...")
        parser.add_option("", "--debug", action="store_true", default=False,
                        help="Launch Tx debugger")
        (options, args) = parser.parse_args ()

        if len(args) != 0:
            parser.print_help()
            sys.exit(1)

        if options.playlist == None:
            print "No playlist specified\n"
            sys.exit()

        # parse playlist
        playlist = read_playlist(options.playlist)

        # setup IQ rate to 320kS/s and audio rate to 32kS/s
        self.u = usrp.sink_c()
        self.dac_rate = self.u.dac_rate() # 128 MS/s
        self.usrp_interp = 400

```

```

self.u.set_interp_rate(self.usrp_interp)
self.usrp_rate = self.dac_rate / self.usrp_interp    # 320 kS/s
self.sw_interp = 10
self.audio_rate = self.usrp_rate / self.sw_interp    # 32 kS/s

# determine the daughterboard subdevice we're using
if options.tx_subdev_spec is None:
    options.tx_subdev_spec = usrp.pick_tx_subdevice(self.u)

m = usrp.determine_tx_mux_value(self.u, options.tx_subdev_spec)
self.u.set_mux(m)

self.subdev = usrp.selected_subdev(self.u, options.tx_subdev_spec)
print "Using TX d'board %s" % (self.subdev.side_and_name(),)

self.subdev.set_gain(self.subdev.gain_range()[1])    # set max Tx
gain

if not self.set_freq(options.freq):
    freq_range = self.subdev.freq_range()
    print "Failed to set frequency to %s. Daughterboard supports %s
to %s" % (
        eng_notation.num_to_str(options.freq),
        eng_notation.num_to_str(freq_range[0]),
        eng_notation.num_to_str(freq_range[1]))
    raise SystemExit
self.subdev.set_enable(True)                        # enable
transmitter
print "TX freq %1.2f MHz\n" % (options.freq/1e6)

gain = gr.multiply_const_cc(4000.0)

# loop through playlist
if options.randomize:
    i = random.randint(0,len(playlist)-1)
else:
    i = 0

self.fg = gr.top_block()
fmtx = blks2.wfm_tx(self.audio_rate, self.usrp_rate,max_dev=75e3,
tau=75e-6)
while 1:

    outputfile = mktempfn()
    # write raw sound to named pipe in background
    thread.start_new_thread(mp3toraw,(playlist[i],outputfile))
    # sleep until we are sure there is something to play
    time.sleep(3)

    print "File size %d\n" % int(os.stat(outputfile)[6])

    src = gr.file_source(gr.sizeof_float, outputfile, False)

    # connect blocks
    self.fg.connect(src, fmtx, gain, self.u)

    print "Starting to play\n"

```

```

self.fg.run()
print "Done..."

# stop and wait to finish
self.fg.stop()
self.fg.wait()
self.fg.disconnect(src, fmtx, gain, self.u)

os.remove(outputfile)
# hack, we should get pid and kill sox only if necessary.
os.system("killall sox")

if options.randomize:
    i = random.randint(0,len(playlist)-1)
else:
    i = (i + 1) % len(playlist)

def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.
    """
    r = self.u.tune(self.subdev.which(), self.subdev, target_freq)
    if r:
        print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
        print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
        print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
        print "r.inverted      =", r.inverted
        return True
    return False

if __name__ == '__main__':
    wfm_tx()

```

Transmitter python script for USRP2:

fmradio_usrp2.py:

```

#!/usr/bin/env python

from gnuradio import gr, eng_notation
from gnuradio import usrp2
from gnuradio import audio
from gnuradio import blks2
from gnuradio.eng_option import eng_option
from optparse import OptionParser
from usrpm import usrp_dbid

import math, re, sys, thread, time, tempfile, os, random

def mp3toraw(filename,outputfile):
    print("nice -n 19 sox -v 5.0 \"%s\" -r 96000 -t raw -f -L -c 1 %s\n" %
(filename,outputfile))

```

```

    os.system("nice -n 19 sox -v 5.0 \"%s\" -r 96000 -t raw -f -L -c 1 %s" %
(filename,outputfile))

# Read in .pls format (can be made e.g., using beep-media-player)
def read_playlist(fname):
    input = open(fname, 'r')

    playlist=[]

    #[playlist]
    l = input.readline()
    # NumberOfEntries
    l = input.readline()
    nentries = int(re.findall("NumberOfEntries=([0-9]+)",l)[0])

    print "Number of items in list %d\n" % nentries
    i = 1
    while l:
        l=input.readline()

        filepath = re.findall("File[0-9]+=(.*)$",l)
        if filepath:
            print filepath[0]
            playlist.append(filepath[0])
            i = i + 1

    input.close()
    return(playlist)

## just create a standard tempfn (sox will create the file, so remove one
made by system)
def mktempfn():
    tf = tempfile.mkstemp(".raw")
    outputfile = tf[1]
    os.close(tf[0])
    os.remove(tf[1])
    return(outputfile)

class wfm_tx:
    def __init__(self):

        parser = OptionParser(option_class=eng_option)
        #parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
        # help="select USRP Tx side A or B")
        parser.add_option("-f", "--freq", type="eng_float", default=90e6,
        help="set Tx frequency to FREQ (default 90e6)",
metavar="FREQ")
        parser.add_option("-l","--playlist", action="store", default=None,
        help="MP3 playlist containing files to air.")
        parser.add_option("-r","--randomize", action="store_true",
default=False,
        help="Randomize playlist...")
        parser.add_option("","--debug", action="store_true", default=False,
        help="Launch Tx debugger")
        (options, args) = parser.parse_args ()

```

```

if len(args) != 0:
    parser.print_help()
    sys.exit(1)

if options.playlist == None:
    print "No playlist specified\n"
    sys.exit()

# parse playlist
playlist = read_playlist(options.playlist)

# setup IQ rate to 320kS/s and audio rate to 32kS/s
self.u = usrp2.sink_32fc()
self.dac_rate = self.u.dac_rate()           # 400 MS/s
self.usrp_interp = 100
self.u.set_interp(self.usrp_interp)
self.usrp_rate = (self.dac_rate / self.usrp_interp) # 4000 kS/s
self.sw_interp = 10
self.audio_rate = self.usrp_rate / self.sw_interp # 400 kHz

# determine the daughterboard subdevice we're using
# if options.tx_subdev_spec is None:
#     options.tx_subdev_spec = usrp.pick_tx_subdevice(self.u)

#m = usrp.determine_tx_mux_value(self.u, options.tx_subdev_spec)
#self.u.set_mux(m)

#self.subdev = usrp.selected_subdev(self.u, options.tx_subdev_spec)
#print "Using TX d'board %s" % (self.subdev.side_and_name(),)

self.u.set_gain(self.u.gain_range()[1]) # set max Tx gain

if not self.set_freq(options.freq):
    freq_range = self.u.freq_range()
    print "Failed to set frequency to %s. Daughterboard supports %s
to %s" % (
        eng_notation.num_to_str(options.freq),
        eng_notation.num_to_str(freq_range[0]),
        eng_notation.num_to_str(freq_range[1]))
    raise SystemExit
#self.u.set_enable(True) # enable transmitter
print "TX freq %1.2f MHz\n" % (options.freq/1e6)

gain = gr.multiply_const_cc(4000.0)

# loop through playlist
if options.randomize:
    i = random.randint(0, len(playlist)-1)
else:
    i = 0

self.fg = gr.top_block()
fmtx = blks2.wfm_tx(self.audio_rate, self.usrp_rate, max_dev=75e3,
tau=75e-6)
while 1:

    outputfile = mktempfn()

```

```

# write raw sound to named pipe in background
thread.start_new_thread(mp3toraw,(playlist[i],outputfile))
# sleep until we are sure there is something to play
time.sleep(1)

print "File size %d\n" % int(os.stat(outputfile)[6])

src = gr.file_source(gr.sizeof_float, outputfile, False)

# connect blocks
self.fg.connect(src, fmtx, gain, self.u)

print "Starting to play\n"
self.fg.run()
print "Done..."

# stop and wait to finish
self.fg.stop()
self.fg.wait()
self.fg.disconnect(src, fmtx, gain, self.u)

os.remove(outputfile)
# hack, we should get pid and kill sox only if necessary.
os.system("killall sox")

if options.randomize:
    i = random.randint(0,len(playlist)-1)
else:
    i = (i + 1) % len(playlist)

def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.
    """
    #r = self.u._real_set_center_freq(self.u, target_freq)
    r = self.u.set_center_freq(target_freq)
    if r:
        print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
        print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
        print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
        print "r.spectrum_inverted  =", r.spectrum_inverted
        return True
    return False

if __name__ == '__main__':
    wfm_tx()

```

Appendix B

Receiver python script for USRP1:

usrp_wfm_rcv.py [6]:

```
#!/usr/bin/env python
#
# Copyright 2005,2006,2007,2009 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING. If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks2
from gnuradio.eng_option import eng_option
from gnuradio.wxgui import slider, powermate
from gnuradio.wxgui import stdgui2, fftsink2, form
from optparse import OptionParser
from usrpm import usrp_dbid
import sys
import math
import wx

def pick_subdevice(u):
    """
    The user didn't specify a subdevice on the command line.
    Try for one of these, in order: TV_RX, BASIC_RX, whatever is on side A.

    @return a subdev_spec
    """
    return usrp.pick_subdev(u, (usrp_dbid.TV_RX,
                               usrp_dbid.TV_RX_REV_2,
                               usrp_dbid.TV_RX_REV_3,
                               usrp_dbid.BASIC_RX))

class wfm_rx_block (stdgui2.std_top_block):
```

```

def __init__(self, frame, panel, vbox, argv):
    stdgui2.std_top_block.__init__(self, frame, panel, vbox, argv)

    parser=OptionParser(option_class=eng_option)
    parser.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                    help="select USRP Rx side A or B (default=A)")
    parser.add_option("-f", "--freq", type="eng_float", default=100.1e6,
                    help="set frequency to FREQ", metavar="FREQ")
    parser.add_option("-g", "--gain", type="eng_float", default=40,
                    help="set gain in dB (default is midpoint)")
    parser.add_option("-V", "--volume", type="eng_float", default=None,
                    help="set volume (default is midpoint)")
    parser.add_option("-O", "--audio-output", type="string", default="",
                    help="pcm device name.  E.g., hw:0,0 or surround51
or /dev/dsp")

    (options, args) = parser.parse_args()
    if len(args) != 0:
        parser.print_help()
        sys.exit(1)

    self.frame = frame
    self.panel = panel

    self.vol = 0
    self.state = "FREQ"
    self.freq = 0

    # build graph

    self.u = usrp.source_c() # usrp is data source

    adc_rate = self.u.adc_rate() # 64 MS/s
    usrp_decim = 200
    self.u.set_decim_rate(usrp_decim)
    usrp_rate = adc_rate / usrp_decim # 320 kS/s
    chanfilt_decim = 1
    demod_rate = usrp_rate / chanfilt_decim
    audio_decimation = 10
    audio_rate = demod_rate / audio_decimation # 32 kHz

    if options.rx_subdev_spec is None:
        options.rx_subdev_spec = pick_subdevice(self.u)

    self.u.set_mux(usrp.determine_rx_mux_value(self.u,
options.rx_subdev_spec))
    self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
    print "Using RX d'board %s" % (self.subdev.side_and_name(),)
    dbid = self.subdev.dbid()
    if not (dbid == usrp_dbid.BASIC_RX or
            dbid == usrp_dbid.TV_RX or
            dbid == usrp_dbid.TV_RX_REV_2 or
            dbid == usrp_dbid.TV_RX_REV_3):
        print "This daughterboard does not cover the required frequency
range"

```

```

        print "for this application. Please use a BasicRX or TVRX
daughterboard."
        raw_input("Press ENTER to continue anyway, or Ctrl-C to exit.")

        chan_filt_coeffs = optfir.low_pass (1,          # gain
                                           usrp_rate,  # sampling rate
                                           80e3,       # passband cutoff
                                           115e3,      # stopband cutoff
                                           0.1,        # passband ripple
                                           60)         # stopband

attenuation
    #print len(chan_filt_coeffs)
    chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)

    self.guts = blks2.wfm_rcv (demod_rate, audio_decimation)

    self.volume_control = gr.multiply_const_ff(self.vol)

    # sound card as final sink
    audio_sink = audio.sink (int (audio_rate),
                             options.audio_output,
                             False) # ok_to_block

    # now wire it all together
    self.connect (self.u, chan_filt, self.guts, self.volume_control,
audio_sink)

    self._build_gui(vbox, usrp_rate, demod_rate, audio_rate)

    if options.gain is None:
        # if no gain was specified, use the mid-point in dB
        g = self.subdev.gain_range()
        options.gain = float(g[0]+g[1])/2

    if options.volume is None:
        g = self.volume_range()
        options.volume = float(g[0]+g[1])/2

    if abs(options.freq) < 1e6:
        options.freq *= 1e6

    # set initial values

    self.set_gain(options.gain)
    self.set_vol(options.volume)
    if not(self.set_freq(options.freq)):
        self._set_status_msg("Failed to set initial frequency")

def _set_status_msg(self, msg, which=0):
    self.frame.GetStatusBar().SetStatusText(msg, which)

def _build_gui(self, vbox, usrp_rate, demod_rate, audio_rate):

    def _form_set_freq(kv):
        return self.set_freq(kv['freq'])

```

```

        if 1:
            self.src_fft = fftsink2.fft_sink_c(self.panel, title="Data from
USRP",
                                                fft_size=512,
sample_rate=usrp_rate,
                                                ref_scale=32768.0, ref_level=0,
y_divs=12)
            self.connect (self.u, self.src_fft)
            vbox.Add (self.src_fft.win, 4, wx.EXPAND)

        if 1:
            post_filt_fft = fftsink2.fft_sink_f(self.panel, title="Post
Demod",
                                                fft_size=1024,
sample_rate=usrp_rate,
                                                y_per_div=10, ref_level=0)
            self.connect (self.guts.fm_demod, post_filt_fft)
            vbox.Add (post_filt_fft.win, 4, wx.EXPAND)

        if 0:
            post_deemph_fft = fftsink2.fft_sink_f(self.panel, title="Post
Deemph",
                                                fft_size=512,
sample_rate=audio_rate,
                                                y_per_div=10, ref_level=-
20)
            self.connect (self.guts.deemph, post_deemph_fft)
            vbox.Add (post_deemph_fft.win, 4, wx.EXPAND)

# control area form at bottom
self.myform = myform = form.form()

hbox = wx.BoxSizer(wx.HORIZONTAL)
hbox.Add((5,0), 0)
myform['freq'] = form.float_field(
    parent=self.panel, sizer=hbox, label="Freq", weight=1,
    callback=myform.check_input_and_call(_form_set_freq,
self._set_status_msg))

hbox.Add((5,0), 0)
myform['freq_slider'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
weight=3,
                                range=(87.9e6, 108.1e6, 0.1e6),
                                callback=self.set_freq)

hbox.Add((5,0), 0)
vbox.Add(hbox, 0, wx.EXPAND)

hbox = wx.BoxSizer(wx.HORIZONTAL)
hbox.Add((5,0), 0)

myform['volume'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Volume",

```

```

weight=3, range=self.volume_range(),
callback=self.set_vol)

hbox.Add((5,0), 1)

myform['gain'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Gain",
                                weight=3,
range=self.subdev.gain_range(),
                                callback=self.set_gain)

hbox.Add((5,0), 0)
vbox.Add(hbox, 0, wx.EXPAND)

try:
    self.knob = powermate.powermate(self.frame)
    self.rot = 0
    powermate.EVT_POWERMATE_ROTATE (self.frame, self.on_rotate)
    powermate.EVT_POWERMATE_BUTTON (self.frame, self.on_button)
except:
    print "FYI: No Powermate or Contour Knob found"

def on_rotate (self, event):
    self.rot += event.delta
    if (self.state == "FREQ"):
        if self.rot >= 3:
            self.set_freq(self.freq + .1e6)
            self.rot -= 3
        elif self.rot <=-3:
            self.set_freq(self.freq - .1e6)
            self.rot += 3
    else:
        step = self.volume_range()[2]
        if self.rot >= 3:
            self.set_vol(self.vol + step)
            self.rot -= 3
        elif self.rot <=-3:
            self.set_vol(self.vol - step)
            self.rot += 3

def on_button (self, event):
    if event.value == 0:          # button up
        return
    self.rot = 0
    if self.state == "FREQ":
        self.state = "VOL"
    else:
        self.state = "FREQ"
    self.update_status_bar ()

def set_vol (self, vol):
    g = self.volume_range()
    self.vol = max(g[0], min(g[1], vol))
    self.volume_control.set_k(10**(self.vol/10))
    self.myform['volume'].set_value(self.vol)
    self.update_status_bar ()

```

```

def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.

    @param target_freq: frequency in Hz
    @rtype: bool

    Tuning is a two step process. First we ask the front-end to
    tune as close to the desired frequency as it can. Then we use
    the result of that operation and our target_frequency to
    determine the value for the digital down converter.
    """
    r = usrp.tune(self.u, 0, self.subdev, target_freq)

    if r:
        self.freq = target_freq
        self.myform['freq'].set_value(target_freq)          # update
displayed value
        self.myform['freq_slider'].set_value(target_freq)  # update
displayed value
        self.update_status_bar()
        self._set_status_msg("OK", 0)
        return True

    self._set_status_msg("Failed", 0)
    return False

def set_gain(self, gain):
    self.myform['gain'].set_value(gain)          # update displayed value
    self.subdev.set_gain(gain)

def update_status_bar (self):
    msg = "Volume:%r Setting:%s" % (self.vol, self.state)
    self._set_status_msg(msg, 1)
    self.src_fft.set_baseband_freq(self.freq)

def volume_range(self):
    return (-20.0, 0.0, 0.5)

if __name__ == '__main__':
    app = stdgui2.stdapp (wfm_rx_block, "USRP WFM RX")
    app.MainLoop ()

```

Receiver python script for USRP2:

usrp2_wfm_rcv.py [6]:

```

#!/usr/bin/env python
#
# Copyright 2005,2006,2007,2008,2009 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#

```

```

# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, gru, eng_notation, optfir
from gnuradio import audio
from gnuradio import usrp2
from gnuradio import blks2
from gnuradio.eng_option import eng_option
from gnuradio.wxgui import slider, powermate
from gnuradio.wxgui import stdgui2, fftsink2, form
from optparse import OptionParser
import sys
import math
import wx

class wfm_rx_block (stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__(self, frame, panel, vbox, argv)

        parser = OptionParser(option_class=eng_option)
        parser.add_option("-e", "--interface", type="string", default="eth0",
            help="select Ethernet interface, default is eth0")
        parser.add_option("-m", "--mac-addr", type="string", default="",
            help="select USRP by MAC address, default is auto-
select")
        #parser.add_option("-A", "--antenna", default=None,
        #    help="select Rx Antenna (only on RFX-series
boards)")
        parser.add_option("-f", "--freq", type="eng_float", default=100.1,
            help="set frequency to FREQ", metavar="FREQ")
        parser.add_option("-g", "--gain", type="eng_float", default=None,
            help="set gain in dB (default is midpoint)")
        parser.add_option("-V", "--volume", type="eng_float", default=None,
            help="set volume (default is midpoint)")
        parser.add_option("-O", "--audio-output", type="string", default="",
            help="pcm device name.  E.g., hw:0,0 or surround51
or /dev/dsp")

        (options, args) = parser.parse_args()
        if len(args) != 0:
            parser.print_help()
            sys.exit(1)

        self.frame = frame

```

```

self.panel = panel

self.vol = 0
self.state = "FREQ"
self.freq = 0

# build graph

self.u = usrp2.source_32fc(options.interface, options.mac_addr)

adc_rate = self.u.adc_rate()           # 100 MS/s
usrp_decim = 312
self.u.set_decim(usrp_decim)
usrp_rate = adc_rate / usrp_decim      # ~320 kS/s
chanfilt_decim = 1
demod_rate = usrp_rate / chanfilt_decim
audio_decimation = 10
audio_rate = demod_rate / audio_decimation # ~32 kHz

#FIXME: need named constants and text descriptions available to (gr-
)usrp2 even
#when usrp(1) module is not built.  A usrp_common module, perhaps?
dbid = self.u.daughterboard_id()
print "Using RX d'board 0x%04X" % (dbid,)
if not (dbid == 0x0001 or #usrp_dbid.BASIC_RX
        dbid == 0x0003 or #usrp_dbid.TV_RX
        dbid == 0x000c or #usrp_dbid.TV_RX_REV_2
        dbid == 0x0040): #usrp_dbid.TV_RX_REV_3
    print "This daughterboard does not cover the required frequency
range"
    print "for this application.  Please use a BasicRX or TVRX
daughterboard."
    raw_input("Press ENTER to continue anyway, or Ctrl-C to exit.")

chan_filt_coeffs = optfir.low_pass (1,           # gain
                                   usrp_rate,    # sampling rate
                                   80e3,         # passband cutoff
                                   115e3,        # stopband cutoff
                                   0.1,          # passband ripple
                                   60)           # stopband
attenuation
#print len(chan_filt_coeffs)
chan_filt = gr.fir_filter_ccf (chanfilt_decim, chan_filt_coeffs)

self.guts = blks2.wfm_rcv (demod_rate, audio_decimation)

self.volume_control = gr.multiply_const_ff(self.vol)

# sound card as final sink
audio_sink = audio.sink (int (audio_rate),
                        options.audio_output,
                        False) # ok_to_block

# now wire it all together
self.connect (self.u, chan_filt, self.guts, self.volume_control,
audio_sink)

```

```

self._build_gui(vbox, usrp_rate, demod_rate, audio_rate)

if options.gain is None:
    # if no gain was specified, use the mid-point in dB
    g = self.u.gain_range()
    options.gain = float(g[0]+g[1])/2

if options.volume is None:
    g = self.volume_range()
    options.volume = float(g[0]+g[1])/2

if abs(options.freq) < 1e6:
    options.freq *= 1e6

# set initial values

self.set_gain(options.gain)
self.set_vol(options.volume)
if not(self.set_freq(options.freq)):
    self._set_status_msg("Failed to set initial frequency")

def _set_status_msg(self, msg, which=0):
    self.frame.GetStatusBar().SetStatusText(msg, which)

def _build_gui(self, vbox, usrp_rate, demod_rate, audio_rate):

    def _form_set_freq(kv):
        return self.set_freq(kv['freq'])

    if 1:
        self.src_fft = fftsink2.fft_sink_c(self.panel, title="Data from
USRP2",
                                         fft_size=512,
sample_rate=usrp_rate,
                                         ref_scale=1.0, ref_level=0, y_divs=12)
        self.connect (self.u, self.src_fft)
        vbox.Add (self.src_fft.win, 4, wx.EXPAND)

    if 1:
        post_filt_fft = fftsink2.fft_sink_f(self.panel, title="Post
Demod",
                                         fft_size=1024,
sample_rate=usrp_rate,
                                         y_per_div=10, ref_level=0)
        self.connect (self.guts.fm_demod, post_filt_fft)
        vbox.Add (post_filt_fft.win, 4, wx.EXPAND)

    if 0:
        post_deemph_fft = fftsink2.fft_sink_f(self.panel, title="Post
Deemph",
                                         fft_size=512,
sample_rate=audio_rate,
                                         y_per_div=10, ref_level=-
20)

```

```

        self.connect (self.guts.deemph, post_deemph_fft)
        vbox.Add (post_deemph_fft.win, 4, wx.EXPAND)

# control area form at bottom
self.myform = myform = form.form()

hbox = wx.BoxSizer(wx.HORIZONTAL)
hbox.Add((5,0), 0)
myform['freq'] = form.float_field(
    parent=self.panel, sizer=hbox, label="Freq", weight=1,
    callback=myform.check_input_and_call(_form_set_freq,
self._set_status_msg))

hbox.Add((5,0), 0)
myform['freq_slider'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
weight=3,
                                range=(87.9e6, 108.1e6, 0.1e6),
                                callback=self.set_freq)

hbox.Add((5,0), 0)
vbox.Add(hbox, 0, wx.EXPAND)

hbox = wx.BoxSizer(wx.HORIZONTAL)
hbox.Add((5,0), 0)

myform['volume'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Volume",
                                weight=3, range=self.volume_range(),
                                callback=self.set_vol)

hbox.Add((5,0), 1)

myform['gain'] = \
    form.quantized_slider_field(parent=self.panel, sizer=hbox,
label="Gain",
                                weight=3, range=self.u.gain_range(),
                                callback=self.set_gain)

hbox.Add((5,0), 0)
vbox.Add(hbox, 0, wx.EXPAND)

try:
    self.knob = powermate.powermate(self.frame)
    self.rot = 0
    powermate.EVT_POWERMATE_ROTATE (self.frame, self.on_rotate)
    powermate.EVT_POWERMATE_BUTTON (self.frame, self.on_button)
except:
    pass
    #print "FYI: No Powermate or Contour Knob found"

def on_rotate (self, event):
    self.rot += event.delta
    if (self.state == "FREQ"):
        if self.rot >= 3:
            self.set_freq(self.freq + .1e6)
            self.rot -= 3

```

```

        elif self.rot <=-3:
            self.set_freq(self.freq - .1e6)
            self.rot += 3
    else:
        step = self.volume_range()[2]
        if self.rot >= 3:
            self.set_vol(self.vol + step)
            self.rot -= 3
        elif self.rot <=-3:
            self.set_vol(self.vol - step)
            self.rot += 3

def on_button (self, event):
    if event.value == 0:          # button up
        return
    self.rot = 0
    if self.state == "FREQ":
        self.state = "VOL"
    else:
        self.state = "FREQ"
    self.update_status_bar ()

def set_vol (self, vol):
    g = self.volume_range()
    self.vol = max(g[0], min(g[1], vol))
    self.volume_control.set_k(10**(self.vol/10))
    self.myform['volume'].set_value(self.vol)
    self.update_status_bar ()

def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.

    @param target_freq: frequency in Hz
    @rypte: bool

    Tuning is a two step process.  First we ask the front-end to
    tune as close to the desired frequency as it can.  Then we use
    the result of that operation and our target_frequency to
    determine the value for the digital down converter.
    """
    r = self.u.set_center_freq(target_freq)
    if r:
        self.freq = target_freq
        self.myform['freq'].set_value(target_freq)          # update
displayed value
        self.myform['freq_slider'].set_value(target_freq)  # update
displayed value
        self.update_status_bar()
        self._set_status_msg("OK", 0)
        return True

    self._set_status_msg("Failed", 0)
    return False

def set_gain(self, gain):

```

```
        self.myform['gain'].set_value(gain)      # update displayed value
        self.u.set_gain(gain)

    def update_status_bar (self):
        msg = "Volume:%r  Setting:%s" % (self.vol, self.state)
        self._set_status_msg(msg, 1)
        self.src_fft.set_baseband_freq(self.freq)

    def volume_range(self):
        return (-20.0, 0.0, 0.5)

if __name__ == '__main__':
    app = stdgui2.stdapp (wfm_rx_block, "USRP2 WFM RX")
    app.MainLoop ()
```