Software Defined Radio

GNU Radio and the USRP

Overview

- What is Software Defined Radio?
- Advantages of Software Defined Radio
- Traditional versus SDR Receivers
- SDR and the USRP
- Using GNU Radio

Introduction

- What is Software Defined Radio (SDR)?
  - Getting code as close to the antenna as possible
  - Replacing hardware with software for modulation/demodulation
- Advantages:
  - Makes communications systems reconfigurable (adapting to new standards)
  - Flexible (universal software device - not special purpose)
  - Filters/Other Hardware
  - Cognitive Radio
Traditional Receiver

- RF Amplifier
- Local Oscillator
- IF Amplifier
- Demodulator

\[ f_{\text{LO}} - f_c \]
\[ f_{\text{LO}} + f_c \]

Traditional vs. SDR Receiver

- Traditional / Hardware Receiver
- SDR Receiver
- Current SDR Receiver
- Future SDR Receiver?

SDR Receiver Using the USRP

- Daughterboard
- Motherboard
- Receiver Front End
- ADC
- FPGA
- USB Controller

Decimation, MUX

GNU Radio software

USRP: Universal Software Radio Peripheral
Quadrature Signal Representation

The received signal, $S(t)$, may be represented as follows:

$$S(t) = I(t)\cos(2\pi f_c t) + Q(t)\sin(2\pi f_c t)$$

- $f_c$ = carrier frequency
- $I(t)$ = in-phase component
- $Q(t)$ = quadrature component

含 amplitude and phase information of baseband signal

• GNU Radio uses $I$ and $Q$ components to demodulate signals
• USRP front end translates the signal to zero frequency and extracts $I$ and $Q$

Extracting $I(t)$ from $S$

$$S(t) = I(t)\cos(2\pi f_c t) + Q(t)\sin(2\pi f_c t)$$

Multiplying both sides by $\cos(2\pi f_c t)$:

$$S(t)\cos(2\pi f_c t) = I(t)\cos^2(2\pi f_c t) + Q(t)\cos(2\pi f_c t)\sin(2\pi f_c t)$$

$$= I(t)\left[\frac{1}{2}(1 + \cos(4\pi f_c t))\right] + Q(t)\left[\frac{1}{2}\sin(4\pi f_c t) + \sin(0)\right]$$

$$= \frac{1}{2}I(t) + \frac{1}{2}I(t)\cos(4\pi f_c t) + \frac{1}{2}Q(t)\sin(4\pi f_c t)$$

Applying this signal to a low pass filter, the output will be:

$$\frac{1}{2}I(t)$$

Extracting $Q(t)$ from $S$

$$S(t) = I(t)\cos(2\pi f_c t) + Q(t)\sin(2\pi f_c t)$$

Multiplying both sides by $\sin(2\pi f_c t)$:

$$S(t)\sin(2\pi f_c t) = I(t)\cos(2\pi f_c t)\sin(2\pi f_c t) + Q(t)\sin^2(2\pi f_c t)$$

$$= I(t)\left[\frac{1}{2}\sin(4\pi f_c t) - \sin(0)\right] + Q(t)\left[\frac{1}{2}(1 - \cos(4\pi f_c t))\right]$$

$$= \frac{1}{2}I(t)\sin(4\pi f_c t) + \frac{1}{2}Q(t)\sin(4\pi f_c t) - \frac{1}{2}I(t)\cos(4\pi f_c t)$$

Applying this signal to a low pass filter, the output will be:

$$\frac{1}{2}Q(t)$$
**USRP Receiver Front End**

- RF Amplifier
- LO
- LPF
- 90°
- LPF

**Analog to Digital Converter (ADC)**

- 12 bit A/D Converter ($2^{12}$ levels)
- 2 volt peak-peak maximum input
- 64 Msamp/second

- Sampling Interval: $\Delta t = \frac{1}{64 \times 10^6} = 0.0156 \mu s$
- Quantization Levels: $\Delta v = \frac{2}{2^{12}} = 0.488 \text{mV}$

**Decimation**

- Original sampling rate is 64Msamp/sec
- Converts a portion of spectrum 32 MHz wide
- Generally we are interested in a narrower portion of the spectrum requiring a lower sampling rate
- USB cannot handle that high data rate
- Occurs in the FPGA of the USRP

- $f_s = 64\text{Msamp/sec}$
- $f_s = 64\text{Msamp/sec}$
- $f_s = 500\text{Ksamp/sec}$

- $\frac{64\text{M}}{500\text{K}} = 128$
SDR Receiver with USRP

![Diagram of SDR Receiver with USRP]

USRP - Motherboard/Daughterboard

![Image of USRP Motherboard and Daughterboard]

GNU Radio Software

- Community-based project started in 1998
- GNU Radio application consists of sources (inputs), sinks (outputs) and transform blocks
- Transform blocks: math, filtering, modulation/demodulation, coding, etc.
- Sources: USRP, audio input, file input, signal generator, ...
- Sinks: USRP, audio output, file output, FFT, oscilloscope, ...
- Blocks written in C++
- Python scripts used to connect blocks and form application
Design of a Receiver

- USRP: Set frequency of local oscillator (receive frequency), gain of amplifier, decimation factor
- GNU Radio application: use Python to specify and connect blocks that perform demodulation and decoding

Example: 400 - 500 MHz NBFM Receiver

- Problem: Receive an audio signal (up to 4 KHz) transmitted at 446 MHz using narrowband FM (NBFM) with a 16 KHz transmission bandwidth

Design Procedure

1. Plan the block diagram of system components
2. Determine block parameters
3. Determine decimation rates
4. Write Python script to specify the blocks and connect them together
Determining the Decimation Factors

Total Decimation factor = 8000 = D_1D_2D_3

64Msamp/sec ➔ 8Ksamp/sec

FPGA Decimation Factor, D_1

- Total Decimation factor = 8000 = D_1D_2D_3
- Maximize the decimation in FPGA
- Maximum decimation factor in FPGA = 256
- Select D_1 = 250 (factor of 8000)
- Output sample rate = 64Ms/s / 250 = 256Ks/s
Channel Filter Specification

- Maximum frequency = 16 KHz  → Reduce sample rate to 32 Ks/s
- 256Ks/s / 32Ks/s  → $D_1 = 8$

FM Demodulator

- Maximum frequency = 4 KHz  → Reduce sample rate to 8 Ks/s
- 32Ks/s / 8Ks/s  → $D_3 = 4$
- FM Demodulator block “extracts” audio signal from FM waveform by operating on I and Q

Complete Application Design

- Total decimation ratio = $250*8*4 = 8000$
- Problem: The audio card requires an input sample rate ≥ 44.1 Ks/s
- Solution: Use a Resampler to increase the output sample rate
**Final Application Design**

- Audio Card requires a sample rate $\geq 44.1$ Ks/sec. Use 48 Ks/sec.
- Modify FM Demodulator to have a decimation factor of 1 (no change)
- Increase the sample rate to 48 Ks/sec with Resampler ($x\frac{3}{2}$)

---

**Implementing the Design**

- Create a Python script to specify and connect the various GNU radio blocks
- Blocks are already written in C++
- USRP parameters are set within Python script
- `#` indicates that the line is a comment
- Refer to nbffm.py script

---

**Setting the USRP Parameters**

- The following code sets the USRP Parameters:

```python
# Set USRP data source
u = usrp.source(samp_rate=2E6)

# Tell USRP what daughter card to use and display name
# (0,0) means side 1, 0th channel - picks daughterboard
u.set_cardsบา countered=0, mode=0)

# Set USRP parameters
u.set_channel(channel(0), subdev='B', subdev_index=2)

# Fetch tunable frequency
u.set_rf_freq(freq)
```
Channel Filter Design

- The following code specifies the channel filter and computes the coefficients:

```plaintext
channel_filter coefficients

-60 dB at 0 Hz
-16 dB at 16 kHz
16 dB at -16 kHz
-9 dB at 9 kHz
8 dB at -9 kHz
-8 dB at 8 kHz
0.1 dB at -8 kHz
0 dB at 0 kHz
```

Channel Filter Creation

- The following code creates the channel filter using the coefficients computed:

```plaintext
% Create the channel filter coefficients
channel_filter_coeffs = [1, -2, 1];

% Create the channel filter object
channel_filter = dsp.ChannelFilter('Numerator', channel_filter_coeffs);
```

FM Demodulator

- The following code creates the FM demodulator.
- The demodulator block also includes a low pass filter:

```plaintext
% Create the FM demodulator
modulator = dsp.FMModulator('NumSymbols', 10000, 'SampleRate', 8000, 'Gain', 1);
```
Resampler

- The following code creates the resampler.
- The resampler decimates and/or interpolates the data to adjust the sample rate.

```python
reset resampler to increase sample rate of 32k to 48k
reset by 3 and divide by 2
ramp = bksf nonlinear_resampler_ff(self, 3, 2)
```

Connecting the Blocks

- The following code connects the blocks:
  - Connect the trough output to the channel filter input
    `self.connect('through')`
  - Connect channel filter to demodulator
    `self.connect('channel', demod)`
    `dpwr = audio.sine(10000)`
  - Connect demod to resampler and resampler to speaker
    `self.connect('demod', ramp)`
    `self.connect('ramp', speaker)`

Or, a single connect statement:

```python
self.connect('through', 'channel', demod, 'ramp', speaker)
```

Final Thoughts

- Demonstration
- Storing/creating data
- Transmitters
- Installing GNU radio
- Questions
- Where do we go from here?