# CuteSDR Technical Manual

## Ver. 1.02

## Mar 25, 2013

**by**
**Moe Wheatley, AE4JY**

www.moetronix.com

**This document describes the technical details of the Qt based SDR Interface Program CuteSDR.**

# Table of Contents



# 1. CuteSDR Overview

## *1.1. Concept*

The CuteSDR project is provided as an example program that implements all the basic functionality of a software defined receiver using the multi OS Qt development platform. All user interface, DSP, and network interface modules are written in C++ and no other libraries are required other than those that are provided by the Qt system.

The goal is for other developers to be able to use this project as a template for developing their own custom applications. The code is modular and reasonably well documented so can also be used as a learning tool or just for the curious.

The design philosophy was to have a minimal featured user interface and concentrate on the DSP aspects of the program. Implementing GUI interfaces is well covered in the literature and also specifically by various Qt resources such as videos and extensive help documentation. The specific DSP structures and algorithms needed to implement basic receiver functionality is not so well understood or explained and is not a normal programming skill taught to most software engineers.

This paper will briefly cover the GUI code but will primarily concentrate on the design decisions and algorithms required to perform basic radio functions. The methods described here are by no means the only or best way to implement a software defined receiver but are functional and reasonably high performance.

## *1.2. Basic Features*

The user interface is a basic GUI design which has a power vs frequency display using a 2D plot as well as a scrolling waterfall type display where color indicates intensity.

A graphical display of current receive frequency and filter bandwidth allows signal tuning and bandwidth adjustment using a mouse, its buttons and scroll wheel.

Radio center frequency, demodulator frequency, as well as various display scaling controls are provided on the main screen. All other setup functions are organized in pull down menus.

Several basic demodulation modes are implemented such as AM, SSB, CW, and FM. A simple UDP discovery scheme allows easy setup and discovery of networked radios.

Only RFSPACE Inc. SDR's are supported using the network interface. www.rfspace.com

## *1.3. Prerequisites*

This document is aimed at a technical audience with a basic knowledge of math and digital signal processing fundamentals. Signal processing is primarily performed using complex data samples so a basic understanding of complex (I/Q) data and operations on complex data is needed. This concept is not that difficult and can be found in any beginning DSP book. The idea of negative frequency may raise some eyebrows, but the idea is no different than thinking of negative velocity such as when your car is moving backwards instead of forwards and describing it as negative speed.

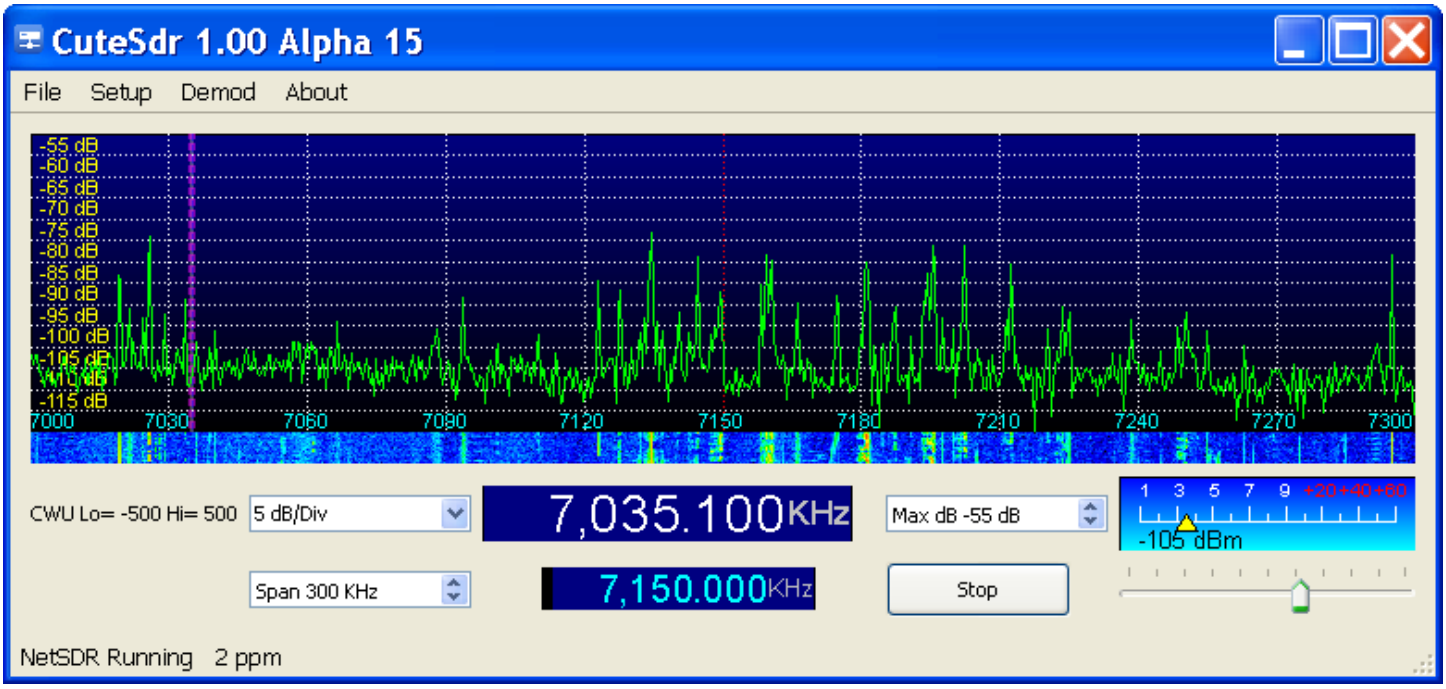
The DSP blocks can be used without much knowledge of how they work except if one wants to modify or add to them.

## *1.4. License*

The source code is released under the Simplified BSD open source license and can be used in open source projects or commercial projects as long as you don't claim the code as your own design. It is hoped that unless a derived work is to be used in a closed source commercial product, that the author will also make their source code for improvements available to the community.

# 2. CuteSDR GUI Controls

## *2.1. Main Controls*




The main screen has a 2D and Waterfall display of the radio spectrum. Two frequency controls are used to vary the radio center frequency and the demodulation frequency.

Three spin controls allow adjusting the screen span, vertical scale, and vertical offset.

A Start/Stop button is used to control the run state of the radio and program.

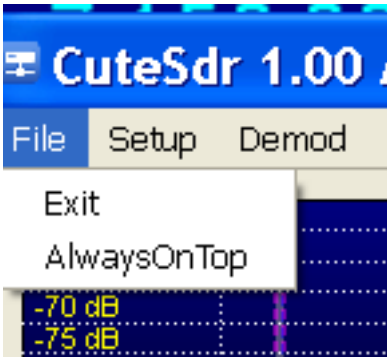
An S-Meter is implemented to display signal strength and a slider is used to adjust audio volume.

All other program control items are accessed through pull down menus from the top menu bar.

## *2.2. Menu Controls*

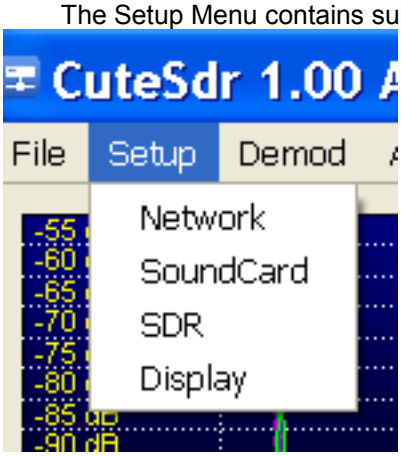
### 2.2.1 File Menu

The File Menu can be used to exit the program or select the “Always On Top” mode to keep on top of other applications.




### 2.2.2 Setup Menu

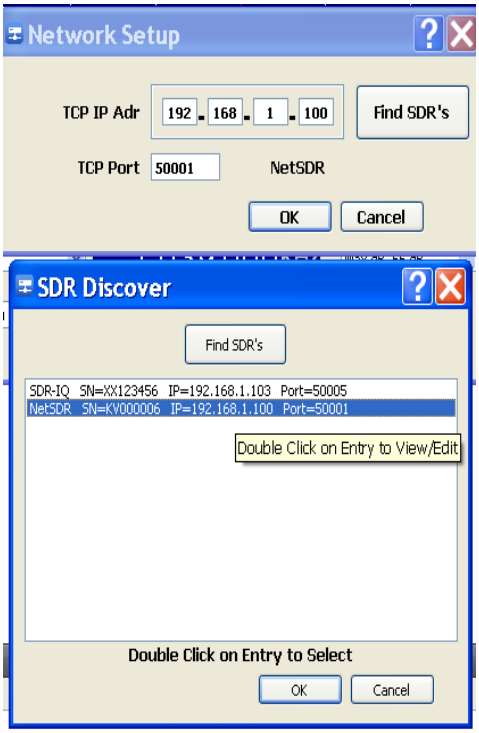
The Setup Menu contains sub menus for setting up various program items.



### 2.2.3 Network Menu

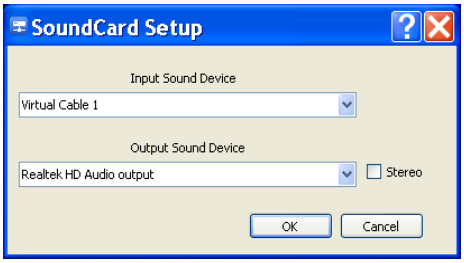
The Network Menu is used to select the IP address and Port number of the desired SDR to connect to. Pressing the “Find SDR's” button invokes an automatic discovery menu that lists all available RFSPACE SDR's that are currently on the network. The desired one can then be selected and its address and port are filled in. Currently the RFSPACE SDR-IP and NetSDR are supported directly.

The SDR-14 and SDR-IQ can be used but require running a small server application(SDRxxServer.exe) on a Windows machine to allow the USB devices to talk to the network. This server application is installed with SpectraVue and is located in the folder where Spectavue.exe is placed.



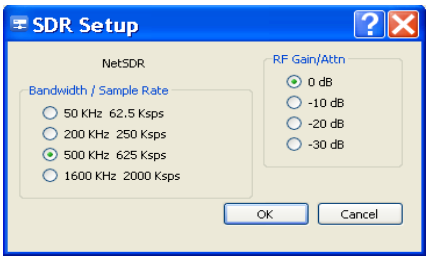

### 2.2.4 Sound Card Menu

This menu is used to select the output sound card for the receiver. The input sound card selection is not used at this time.



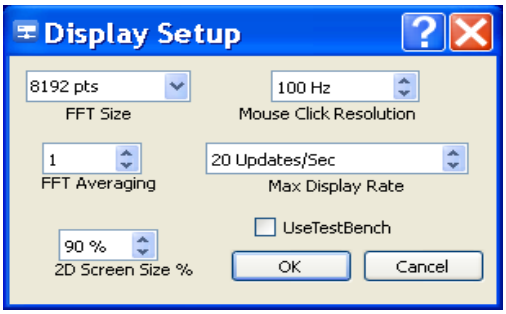

### 2.2.5 SDR Menu

This menu allows one to select the radio sample rate/user bandwidth and RF attenuation. The available sample rates and bandwidths are dependent upon which radio is connected.



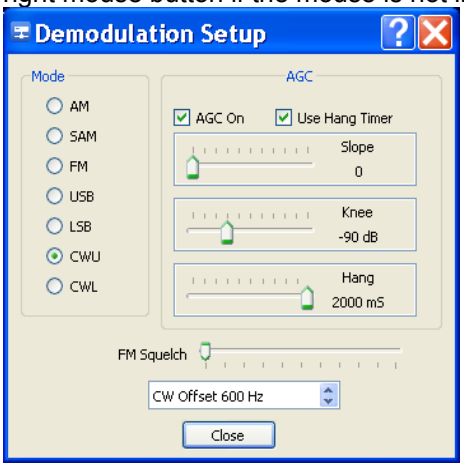

### 2.2.6 Display Menu

The Display Menu provides user control of various display and GUI settings.
The FFT size can be selected for the display.
The Mouse Click Resolution sets the coarseness of selecting a frequency on the display with the mouse.
The FFT averaging can be used to smooth out the display.
The Display update rate can be adjusted. Use a slower update rate on slower PC's.
The percentage of the screen used for the 2D display can be selected.
A “TestBench” screen can be displayed that is useful for debugging DSP code and provides a simple sweep generator, a noise generator, and a pulsed sine generator. The signal stream can be displayed from various points in the DSP signal chain and view in frequency or time domain with a basic triggered scope view.




### 2.2.7 Demod Menu

The Demod menu is used to select the various demodulation parameters. This menu can also be invoked using the right mouse button if the mouse is not in the display screen area.



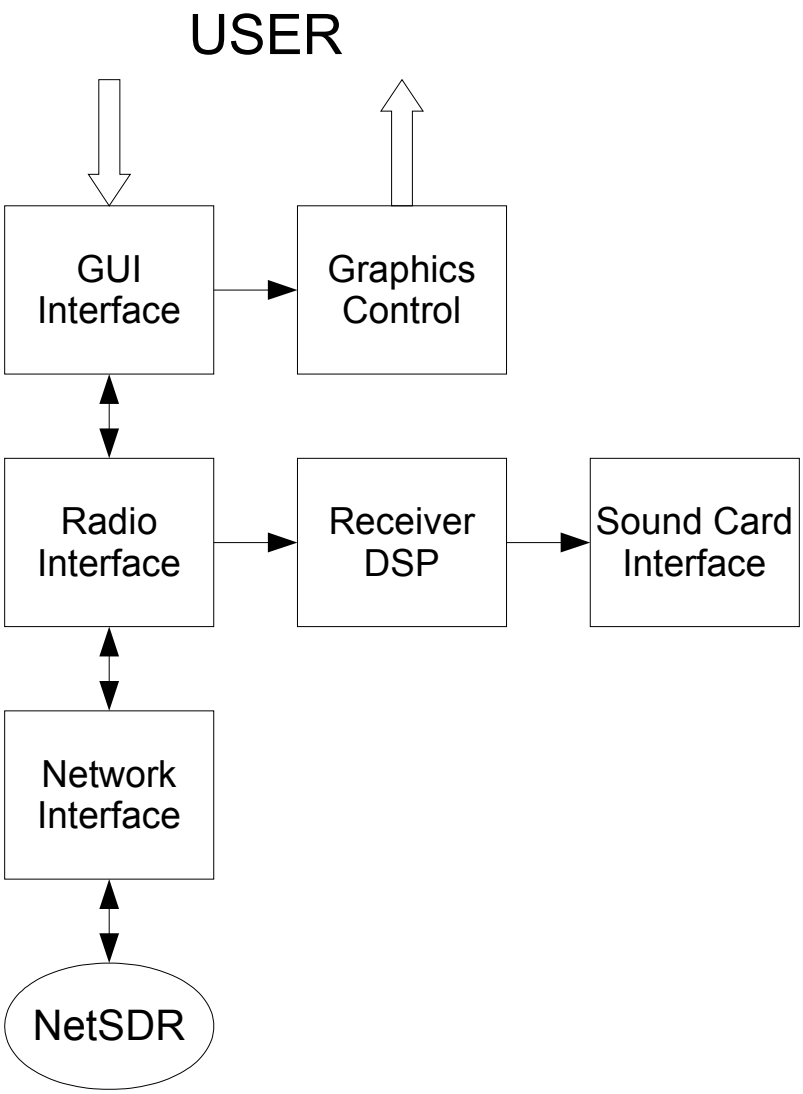

### 2.2.8 About Menu

The About Menu displays program version and other program information.

# 3. CuteSDR Software Architecture

## *3.1. Birds Eye View*

The following diagram shows the high level modular view of CuteSDR:




The user interface controls all the interaction with the user for setup and program operation.

The graphics control module creates and displays the signal graphs and plots as well as the mouse input controls for the demodulation frequency and filter parameters.

The radio control module acts as a data “traffic cop” and parses and distributes data and status from the radio as well as sends control messages to the radio.

The receiver DSP module performs decimation, filtering, and demodulation functions on the I/Q data stream from the receiver.

The sound card interface manages the synchronization and buffering of demodulated data before being output to the PC sound card.

The network interface manages the TCP control socket and the UDP I/Q data stream to and from the radio.

## *3.2. Software Class Summary*

The following section lists all the C++ classes of CuteSDR and a brief description of their function. More detail will be given to the DSP classes in later sections.

### 3.2.1 Display, Dialogs, Controls

MainWindow
: This module is the top level Qt GUI class that manages overall program functionality such as menu control and saving and recalling persistent settings. This module is the first Qt module to be initialized from main() and from it is spawned all other modules and threads.

CPlotter
: Frame Widget for primary FFT signal display, frequency tuning, and filter bandwidth adjustment.

CDemodSetupDlg
: Pop-up menu for changing demodulator and AGC settings.

CDisplayDlg
: Pop-up menu for changing various display parameters.

CEditNetDlg
: Pop-up menu for changing network parameters.

CSdrDiscoverDlg
: Pop-up menu for discovering and selecting a network connected RFSPACE radio.

CSdrSetupDlg
: Pop-up menu for changing various SDR parameters.

CNoiseProcDlg
: Pop-up menu for changing noise processing parameters.

CSoundDlg
: Pop-up menu for selecting the sound card

CIPEditWidget
: Control widget for editing an IP address.

CSliderctrl
: Control widget for slider control with text.

CFreqCtrl
: Control widget for displaying and editing frequency.

CMeter
: Control widget for displaying signal strength.

CTestBench
: Control widget for Viewing internal signals and generating test data.

CAboutDlg
: Pop-up menu for displaying program version and general information.

CRdsDecode
: Performs basic decoding of RDS messages for display.

### 3.2.2 DSP Functions

CFft
: Class that implements an FFT for use in displaying power versus frequency.

CDownConvert
: Class to perform base band conversion of complex I/Q data and also sample rate decimation

CFastFIR
: Class to implement fast convolution filtering of complex I/Q data.

CFir
: Class to implement a FIR filter with various design options.

CIir
: Class to implement a IIR filter with various design options.

CFractResampler

Class to implement a fractional resampler for rate matching to the sound card.

CAgc

Class to implement automatic gain control.

CAmDemod

Class to implement AM demod.

CSamDemod

Class to implement Synchronous AM demod.

CDemodulator

Main class that manages all the demodulator functions.

CFmDemod

Class that performs narrow band FM demodulation.

CWFmDemod

Class that performs wide band FM demodulation and Stereo and RBDS broadcast decoding.

CWFmMod

Class that performs wide band FM modulation, Stereo, and RBDS encoding for testing.

CNoiseProc

Class that performs noise processing.

CSMeter

Class that derives the S-Meter data from the data stream for upper level GUI display.

### 3.2.3 Interface

CAscpMsg

Helper class to format radio messages to go to the radio and decompose messages from the radio.

CNetIOBase

Network base class that provides low level network interface and message assembly and queuing.

CUdp

Thread used by CNetIOBase class for managing low level UDP network data.

CDataProcess

Thread used by CSdrInterface class for removing I/Q data from the FIFO and calling a derived function to process the data for display and demodulation.

CSdrInterface

Class derived from CNetIOBase class to provide radio specific message parsing and creating and sending radio commands. This is the primary interface between the GUI and the radio.

CSoundOut

Class to process and synchronize output data from the demodulator and the sound card.

Cad6620

Helper class to create the msgs to load the filters in the SDR-IQ and SDR-14 radios.

# 4. Technical Description

This section provides a detailed description of the major code modules.

## *4.1. MainWindow*

The main entry point into the program and top level GUI interface is in the MainWindow class. Here all the GUI signals and slots are managed as well as implementing a persistent data mechanism for saving/recalling program settings between program sessions.

## *4.2. GUI Interface*

The GUI interface uses standard Qt objects which are described in the Qt documentation and many other third party videos and tutorials so will not be discussed here. The Qt Creator forms editor was used to manage all the GUI objects.

The main graphics screens are implemented by the Cplotter class. There are three Qpixmap objects that are used to display the 2D spectrum and waterfall displays.

```
QPixmap m_2DPixmap;
QPixmap m_OverlayPixmap;
QPixmap m_WaterfallPixmap;
```

The WaterfallPixmap bitmap object creates a waterfall by scrolling down one line with the scroll(...) member function then drawing the new line at the top of the bitmap.

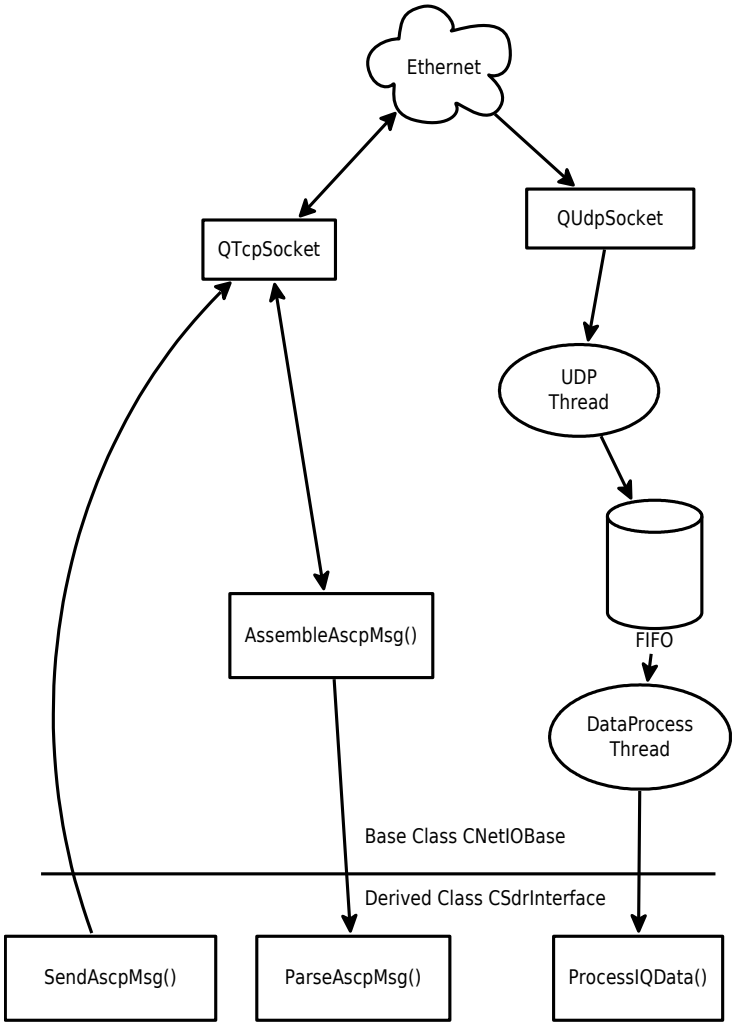
```
m_WaterfallPixmap.scroll(0,1,0,0, w, h);
```

The 2D display has two pixmaps. One is an overlay that contains the grid, text, etc. information that does not change with each screen update. To update the screen, this m_OverlayPixmap is copied into the m_2DPixmap then the 2D spectrum data is drawn on top of that. This is faster since all the overlay graphics need only be created once.

```
m_2DPixmap = m_OverlayPixmap.copy(0,0,w,h);
```

### *4.3. Network Interface*

The network interface consists of two independent data paths. A TCP socket connection between the radio and PC is used for all the command and status messages. A separate UDP port is used just for the high speed I/Q data. This scheme provides a guaranteed connection for command and control while allowing the much more efficient UDP data transfer method for the I/Q data stream. The design decision for this was to have a reliable connection to the radio that could even be routed over the Internet for control and things like code updates etc. The UDP data channel allows much higher data throughput over wireless networks where the loss of a few data packets does not affect performance. In reality, on a small wired network like in your home, the UDP data stream very rarely drops a packet.




In CuteSDR, the Class CNetIOBase implements the low level thread that manage the UDP, and the DataProcess thread that takes IQ Data from the FIFO and processes it. The TCP is managed using the normal GUI thread since it is a low bandwidth channel.

The CNetIOBase class only cares about the message length and assembles a complete message. It then calls the derived classes methods to actually decode the messages from the radios. This keeps the class independent of the type of radio that is connected and all the specifics are implemented in the derived class.

### 4.3.1 General Protocol Description

The RFSPACE radios use a simple protocol called ASCP(Amateur Station Control Protocol) that consists of a 2 byte header followed by the message data. A “Control Item” is a 16 bit code representing the object that the message is controlling or getting status from such as frequency, attenuation, bandwidth, etc. Data messages do not have the Control Item field and transport data such as I/Q data.

Most control items are common to all RFSPACE radios but there are differences so one must look at the individual protocol specifications when implementing an interface.

The basic message structure starts with a 16 bit header that contains the length of the block in bytes and also a 3 bit type field. If the message is a Control Item, then a 16 bit Control Item code follows the header and contains the code describing the object of the message block. This is followed by an optional number of parameter or data bytes associated with this message. The byte order for all fields greater than 8 bits is "Little Endian" or least significant byte first.

Control Item Message block format:

| 16 bit Header(lsb msb) | 16 bit Control Item(lsb msb) | Parameter Bytes |
|---|---|---|

Data Item Message block format:

| 16 bit Header(lsb msb) | N-Data Bytes |
|---|---|

The 16 bit header is defined as follows:

| 8 bit Length lsb | 3 bit type | 5 bit Length msb |
|---|---|---|

The 13 bit Length parameter value is the total number of bytes in the message including this header. The range of the message Length is 0 to 8191 bytes.

A special case for Data Items is that a message length of Zero is used to specify an actual message length of 8194 bytes(8192 data bytes + 2 header bytes). This allows data blocks of a power of 2 to be used which is useful in dealing with FFT data.

The message type field is used by the receiving side to determine how to process this message block. It has a different meaning depending upon whether the message is from the Host or Target.

| 3 bit Msg Type field | Message Source | Message Type |
|---|---|---|
| 000 | Host | Set Control Item |
| 001 | Host | Request Current Control Item |
| 010 | Host | Request Control Item Range |
| 011 | Host | Data Item ACK from Host to Target |
| 100 | Host | Host Data Item 0 |
| 101 | Host | Host Data Item 1 |
| 110 | Host | Host Data Item 2 |
| 111 | Host | Host Data Item 3 |
| 000 | Target | Response to Set or Request Current Control Item |
| 001 | Target | Unsolicited Control Item |
| 010 | Target | Response to Request Control Item Range |
| 011 | Target | Data Item ACK from Target to Host |
| 100 | Target | Target Data Item 0 |
| 101 | Target | Target Data Item 1 |
| 110 | Target | Target Data Item 2 |
| 111 | Target | Target Data Item 3 |

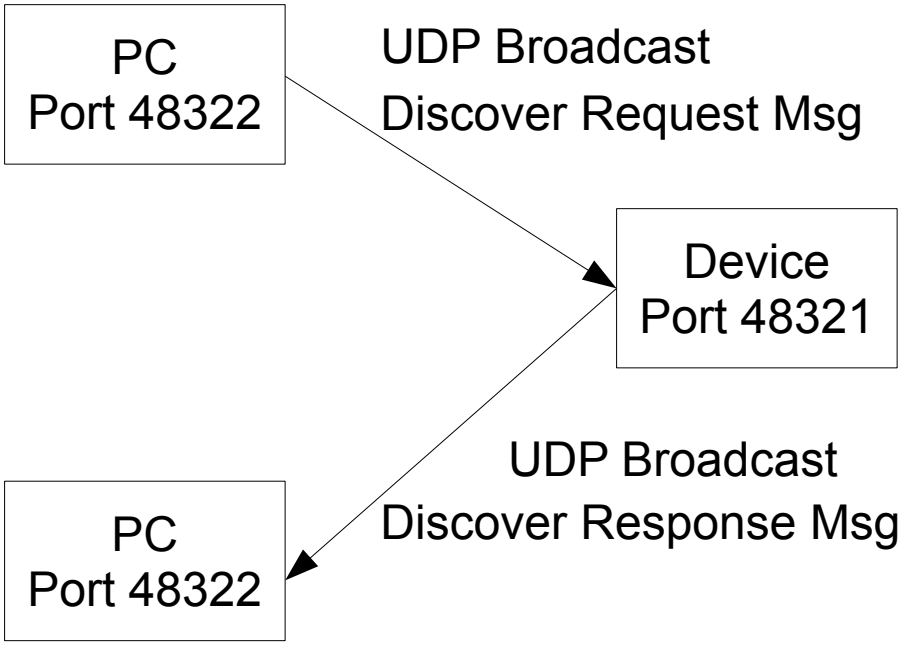
  

### 4.3.2 SNDP Simple Network Discovery Protocol (CSdrDiscoverDlg Class)

The network connected RFSPACE radios implement a simple discovery protocol that uses a UDP broadcast message protocol similar to a DHCP transaction to find and display any connected network radio. This protocol is described in depth in documentation on SourceForge where there is also an open source utility that uses the discover protocol.

http://sourceforge.net/projects/sdrnetsetup/files/doc/

TheCSdrDiscoverDlg class implements a basic GUI that searches for all RFSPACE radios and provides a list of them which the user can then select from.




The sequence of events are as follows for obtaining a devices parameters:

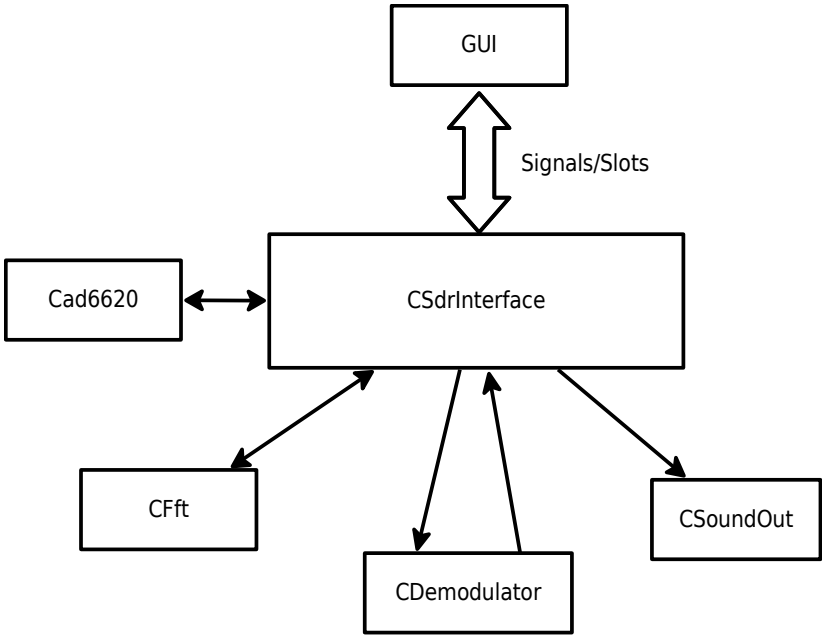
1. PC sends a UDP broadcast Request Message to a device that is listening on Port 48321.
2. The device then sends a UDP broadcast Response Message to the PC which is listening on Port 48322. The response message contains its pertinent network parameters.
3.

The message packet contains a fixed 56 byte field followed by a variable length custom field that is specific to a particular device.

```
 struct DISCOVER_MSG
{               //fixed common 56 byte fields
    unsigned char length[2];        //length of total message in bytes (little endian byte order)
    unsigned char key[2];           //fixed key key[0]==0x5A  key[1]==0xA5
    unsigned char op;               //0==Request(to device)  1==Response(from device) 2 ==Set(to device)
    char name[16];                  //Device name string null terminated
    char sn[16];                    //Serial number string null terminated
    unsigned char ipaddr[16];       //device IP address (little endian byte order)
    unsigned char port[2];          //device Port number (little endian byte order)
    unsigned char customfield;      //Specifies a custom data field for a particular device
..
               //start of optional variable custom byte fields
    unsigned char Custom[N];
}
```

### *4.4. Radio Interface (CSdrInterface Class)*

The CSdrInterface class is the derived class that provides all the specific radio communication and is the interface between the GUI and the radio. It communicates up to the GUI using the normal Qt signal/slot methods.




The ProcessIQData() method performs the top level I/Q data processing by generating the FFT display data, calling the demodulator object, and calling the sound card output object. A secondary operation is provided for the SDR-IQ and SDR-14 receivers that reduces the "NCO Spur" which is a spur that shows up at the center of the screen due to quantization errors of the 16 bit integer data streams. By taking a long DC average of the I and Q data steams, the offset can be subtracted out of the data stream.

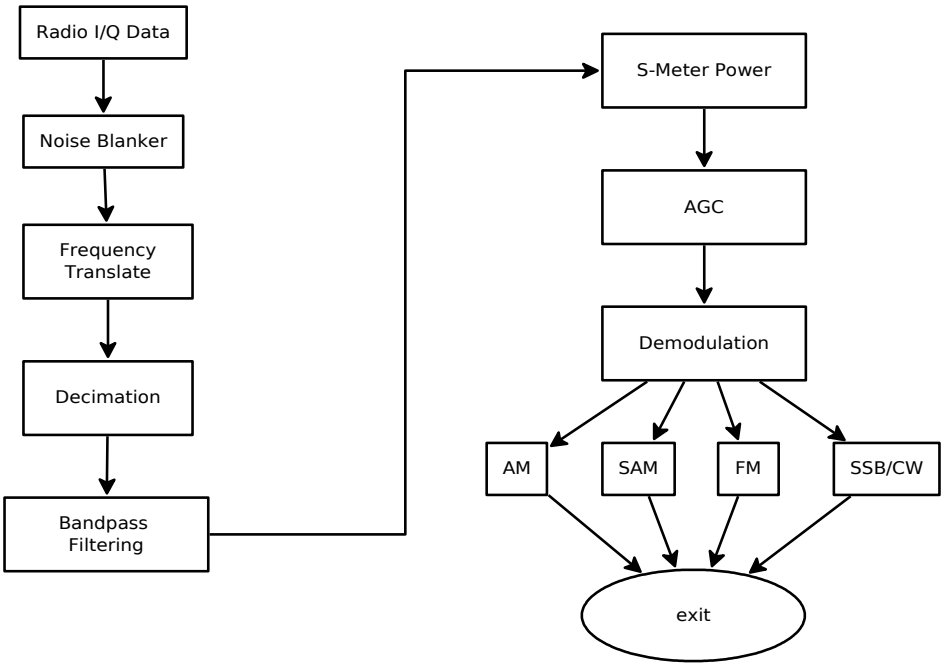
A helper class, "Cad6620" is used to initialize the SDR-IQ and SDR-14 radios which need special hardware initialization for each sample rate used.

Look-up tables are used to convert the GUI selection indexes into radio sample rates and also useable bandwidth which are different with each radio supported.

### 4.4.1 DSP Modules

The following descriptions detail the various C++ classes that implement various DSP functions required to process and demodulate the I/Q data stream.




The raw data that comes from the RFSPACE Radios is formatted as integer I and Q data with data widths of either 16 or 24 bits. The low level I/O routines convert the integer data to floating point scaled data where the maximum magnitude is +-32767.0. The file datatypes.h defines several data types and structures to use in processing the I/Q complex data.

```
TYPEREAL        //Real floating point data
TYPECPX         //Complex floating point data
TYPESTEREO16    //2 Channel 16 bit integer data
TYPEMONO16      //1 Channel 16 bit integer data
```

The underlying floating point type can be set as single or double precision but for CuteSDR it is set as double.

The complex data type “TYPECPX” and “TYPESTEREO16” are structures with member elements .re and .im.

The RFSPACE radio I/Q data is base band complex data that has a “zero” frequency equal to the receiver's NCO center frequency and a bandwidth that is the output sample rate of the radio. The radios can be tuned in 1 Hz steps from 0 to around 30 MHz depending on the model. The sample rates range from 50KHz to 2MHz depending on the model. CuteSDR only supports a handful of sample rates to keep things simple.

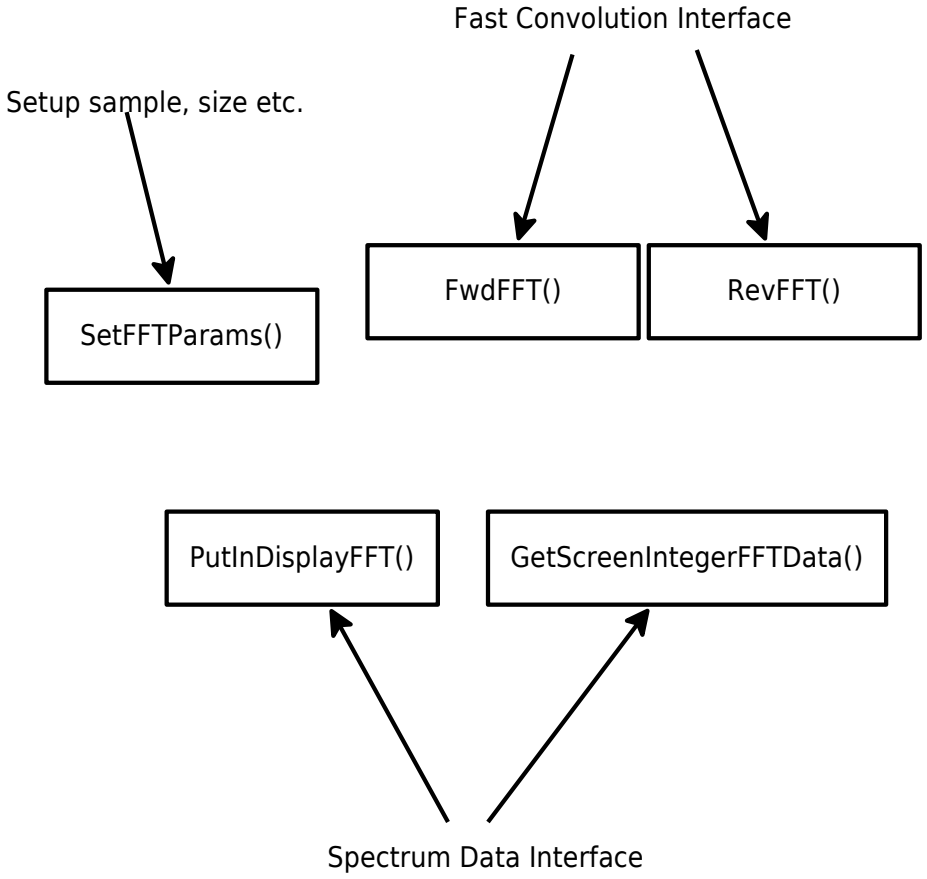
Most DSP classes implement their main data processing function with the same format parameter list. These functions can be overloaded if the data is complex or real. They return the number of processed samples.

“int ProcessData(NumberOfSamples, PointerToInBuffer, PointerToOutBuffer)”

### *4.5. FFT (Cfft Class)*

Th Cfft class implements a fast Fourier transform and several methods to help in creating data formatted for use in a spectrum display. The low level FFT routines were written by Takuya Ooura and a C++ wrapper was put around it as well as some extra methods for creating power spectral data.



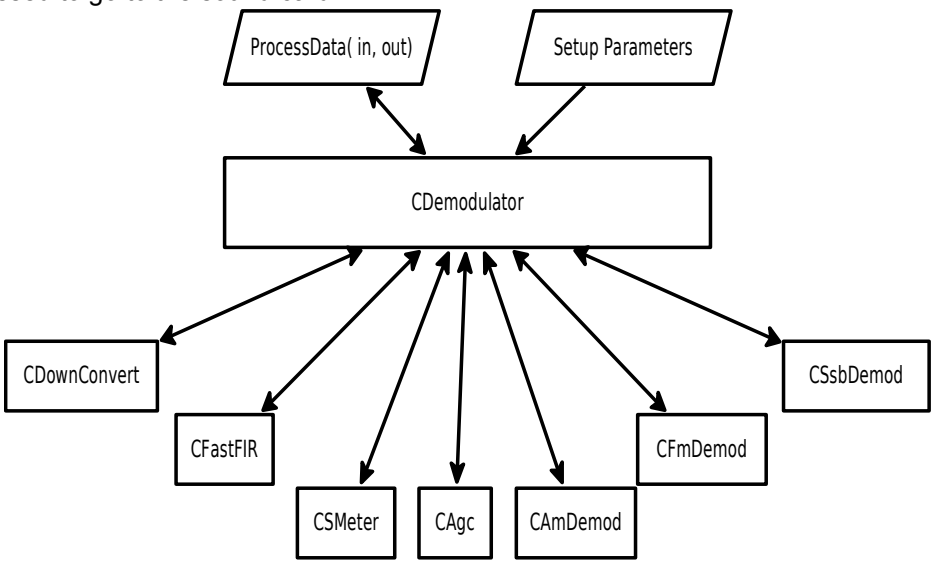

The FwdFFT() and RevFFT() methods provide a raw FFT interface for performing fast convolution which is used in the main DSP filter.

The PutInDisplayFFT() and GetScreenIntegerFFTData() are used to create formatted spectrum data that is easy to use in displaying power spectrum. Display parameters such as screen size in pixels, frequency range, and amplitude range in dB are used to format the data for easy display.

### *4.6. CDemodulator Class*

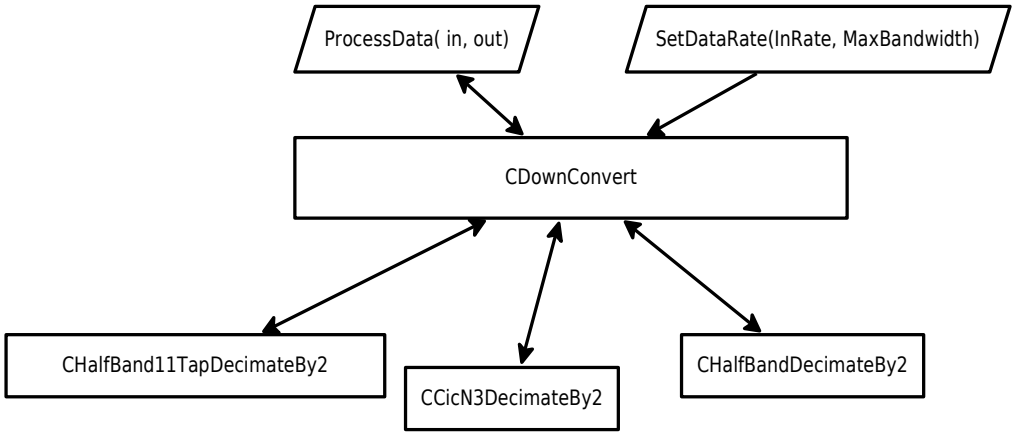
The CDemodulator class encapsulates all the demodulator functions and takes the raw I/Q data from the radio and outputs data processed to go to the sound card.



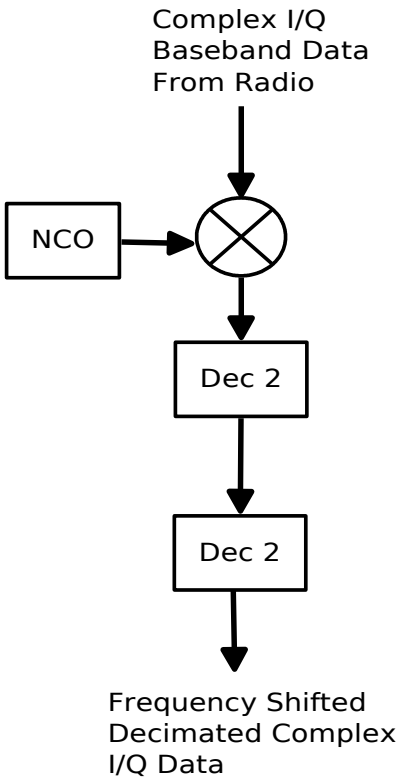

The following code segment performs all the demodulator processing steps:

```
//perform baseband tuning and decimation
int n = m_DownConvert.ProcessData(m_InBufPos, m_pDemodInBuf, m_pDemodInBuf);
//perform main bandpass filtering
n = m_FastFIR.ProcessData (n, m_pDemodInBuf, m_pDemodTmpBuf);
//perform S-Meter processing
m_SMeter.ProcessData(n, m_pDemodTmpBuf, m_OutputRate);
//perform AGC
m_Agc.ProcessData (n, m_pDemodTmpBuf, m_pDemodTmpBuf );
//perform the desired demod action
switch(m_DemodMode)
{
	case DEMOD_AM:
		n = m_pAmDemod->ProcessData(n, m_pDemodTmpBuf, pOutData );
		break;
	case DEMOD_SAM:
		n = m_pSamDemod->ProcessData(n, m_pDemodTmpBuf, pOutData );
		break;
	case DEMOD_FM:
		n = m_pFmDemod->ProcessData(n, m_DemodInfo.HiCut, m_pDemodTmpBuf, pOutData );
		break;
	case DEMOD_USB:
	case DEMOD_LSB:
	case DEMOD_CWU:
	case DEMOD_CWL:
		n = m_pSsbDemod->ProcessData(n, m_pDemodTmpBuf, pOutData);
		break;
}
```

### *4.7. CDownConvert Class*



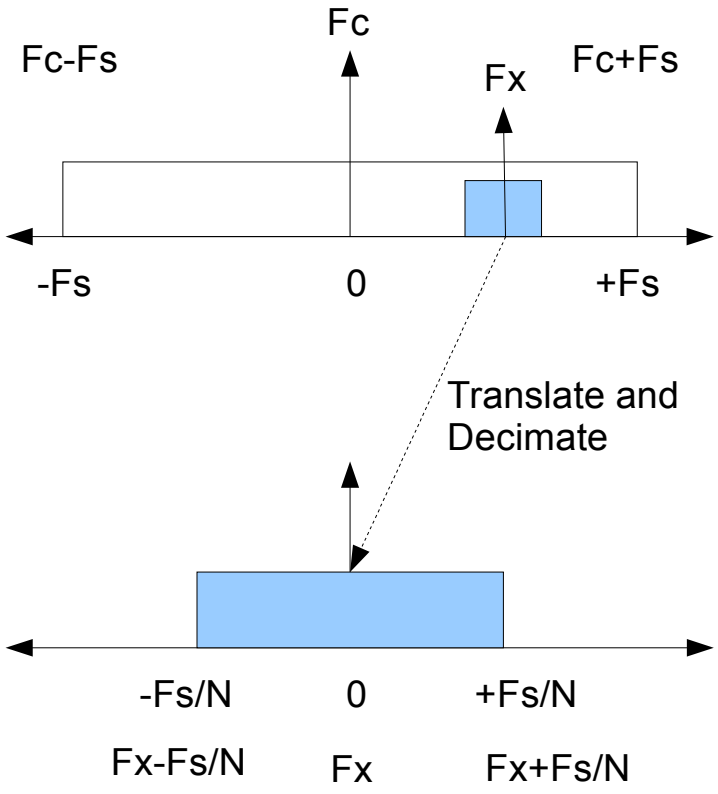

The CDownConvert class performs two functions, frequency translation and decimation.



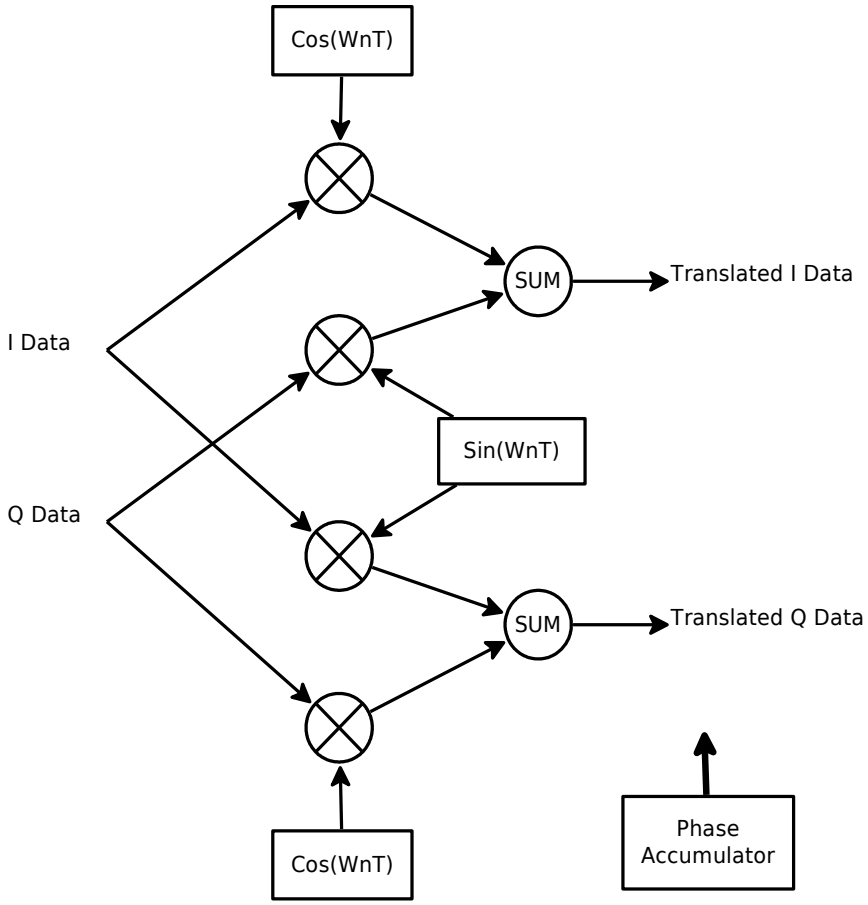

  

### 4.7.1 Frequency Translation

The input signal is first shifted within the receiver pass band by performing a complex frequency translation. This is useful in order to tune to signals within the bandwidth without having to re-tune the radio center frequency which would shift the waterfall display and change the position of all the other signals.




The frequency translation is done by creating a complex software NCO and complex multiplying each input sample by the real and imaginary parts of the NCO generator. This requires four multiplies and 2 adds. More importantly, it requires calculating a new sin and cos for every input sample. Since this runs at the highest sample rate in the system, it is important to make this process as efficient as possible.




In the CDownConvert class, there are three methods implemented to calculate the sin and cos. They are "#ifdef'ed"' so one can compile using any one of the three methods to see which is best.

Method #1: Use standard C library function calls.

```
Osc.re = cos(m_NcoTime);
Osc.im = sin(m_NcoTime);
m_NcoTime += m_NcoInc;
```

Method #2: Call Intel Pentium assembly language instruction that calculates sin and cos in one instruction.

```
asm volatile ("fsincos" : "=%&t" (dASMCos), "=%&u" (dASMSin) : "0" (dPhaseAcc));
   dPhaseAcc += m_NcoInc;
   Osc.re = dASMCos;
   Osc.im = dASMSin;
```

Method #3: Create a software “Quadrature Oscillator” that generates the sin and cos signals.

```
//Called once to set the NCO frequency:
    static  TYPECPX m_Osc1;
    m_NcoInc = K_2PI*m_NcoFreq/m_InRate;
    m_OscCos = cos(m_NcoInc);
    m_OscSin = sin(m_NcoInc);
    m_Osc1.re = 1.0;          //initialize unit vector that will get rotated
    m_Osc1.im = 0.0;


//Called every sample time:
    TYPEREAL OscGn;
    Osc.re = m_Osc1.re * m_OscCos - m_Osc1.im * m_OscSin;
    Osc.im = m_Osc1.im * m_OscCos + m_Osc1.re * m_OscSin;
    OscGn = 1.95 - (m_Osc1.re*m_Osc1.re + m_Osc1.im*m_Osc1.im);
    m_Osc1.re = OscGn * Osc.re;
    m_Osc1.im = OscGn * Osc.im;
```

This method takes a complex unit vector and rotates it every sample time by multiplying the last vector by $e^{(j\Theta)}$ .

Euler's Formula is one of the more important formulas to know as it gives a way to convert between the complex exponential notation and the more implementable quadrature I/Q notation.

$$e^{jx} = \cos(x) + j\sin(x)$$

Using Euler's formula and the fact that Theta is a constant for a fixed frequency, the calculations are simply multiplies and additions.

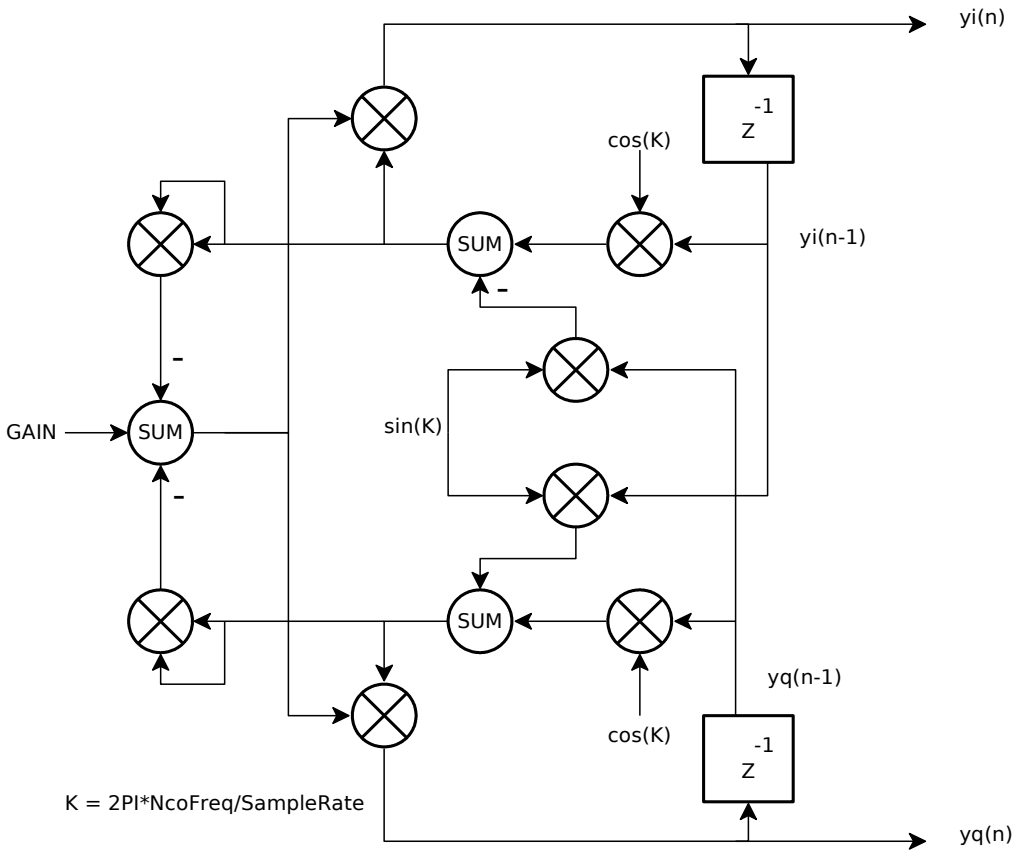
$$y_i(n) = y_i(n-1)\cos(\Theta) - y_q(n-1)\sin(\Theta)$$

$$y_q(n) = y_i(n-1)\sin(\Theta) + y_q(n-1)\cos(\Theta)$$

In order to keep the amplitude from changing due to round off errors, a simple agc calculation is done. A block diagram of the process is shown below.

This method is from Richard Lyons book “Understanding Digital Signal Processing” where he derives and explains the quadrature oscillator in detail. The constants cos(K) and sin(K) are calculated once every time the frequency is changed.

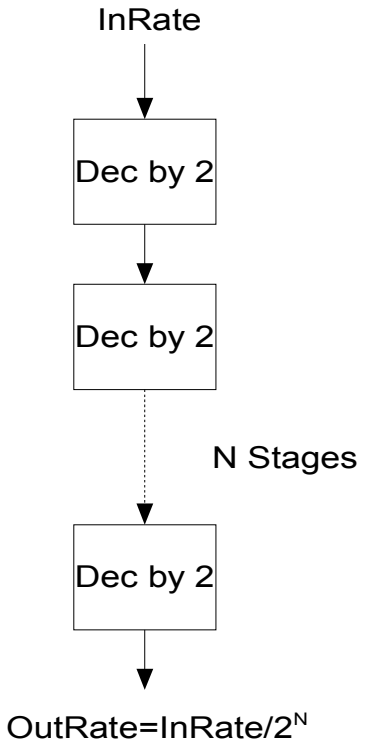



Preliminary tests show that the quadrature oscillator is the fastest of the three methods. There does not seem to be any degradation in signal purity as long as double precision math is used.

#### 4.7.2 Decimation Stages

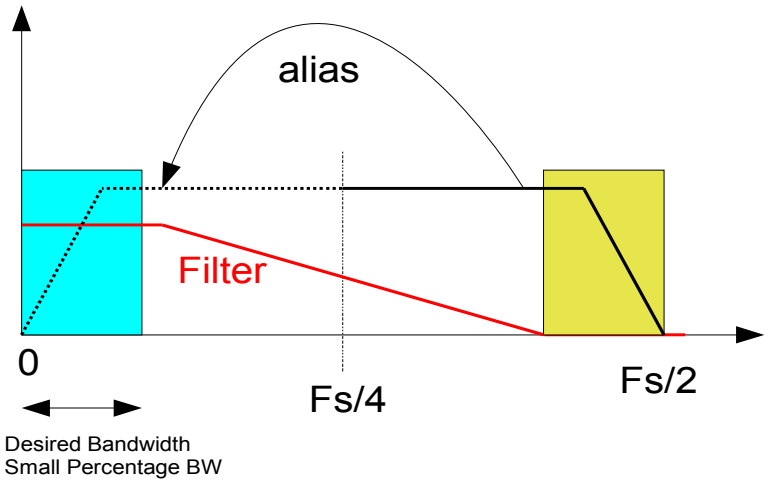
The next step after frequency translation is to decimate the signal to a lower sample rate before performing further processing. This will reduce the processing load. The scheme used by CuteSDR is to run multiple stages that each decimate by a factor of 2 until a sample rate is achieved that is just above the rate needed to support the desired signal bandwidth. Since it is dividing by 2, the worst case rate would be twice what was actually needed.

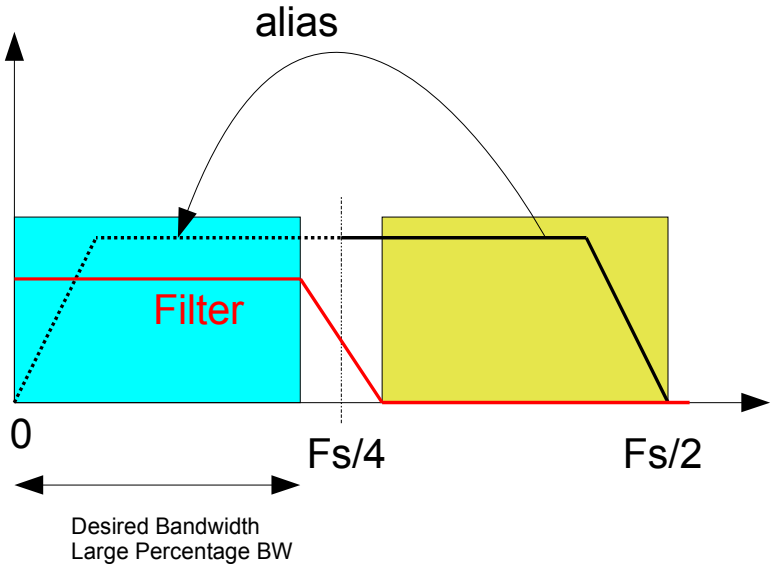



Each decimate by 2 stage has to filter out all everything that can alias into the desired pass band before taking every other sample as output.

The key filter requirement for each decimation stage is the alias free bandwidth after decimation. If the final required bandwidth is 3KHz for example, and the input sample rate is 1Mhz, then the alias filter only needs to protect the 3KHz final bandwidth. This greatly relaxes the filter requirements of the faster rate stages. As the sample rate is reduced, the filter bandwidths get more critical since the desired bandwidth is a greater percentage of the total sample rate bandwidth.



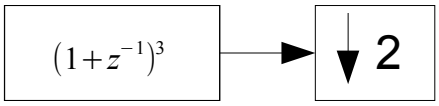

The system design goals for the decimation stages is to have -140dB alias rejection so this sets the limits each kind of filter can support given the sample rate and desired final bandwidth.

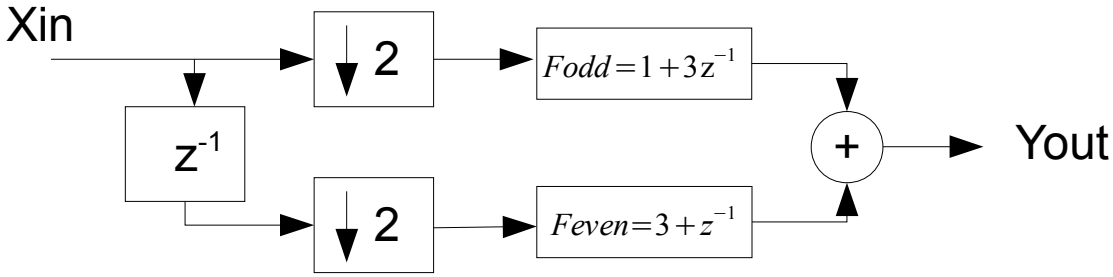
Two different methods of decimating by 2 are used. One is using a CIC (Cascaded Integrator Comb) filter/decimator stage. Normally this type filter is used in hardware decimation stages as it does not require multipliers. They are not normally used in floating point applications since the integrator stage depends on 2's complement numbers to prevent overflow.

Again from Richard Lyons book “Understanding Digital Signal Processing”, a method called polynomial factoring can be used to create filters with the same transfer function but without the integration recursive feedback terms. A 3rd order CIC filter is implemented with this method. Odd and even samples are processed separately and combined for the decimated output.

$$H_{cic3}(z)=(1+z^{-1})^3=1+3z^{-1}+3z^{-2}+z^{-3}$$



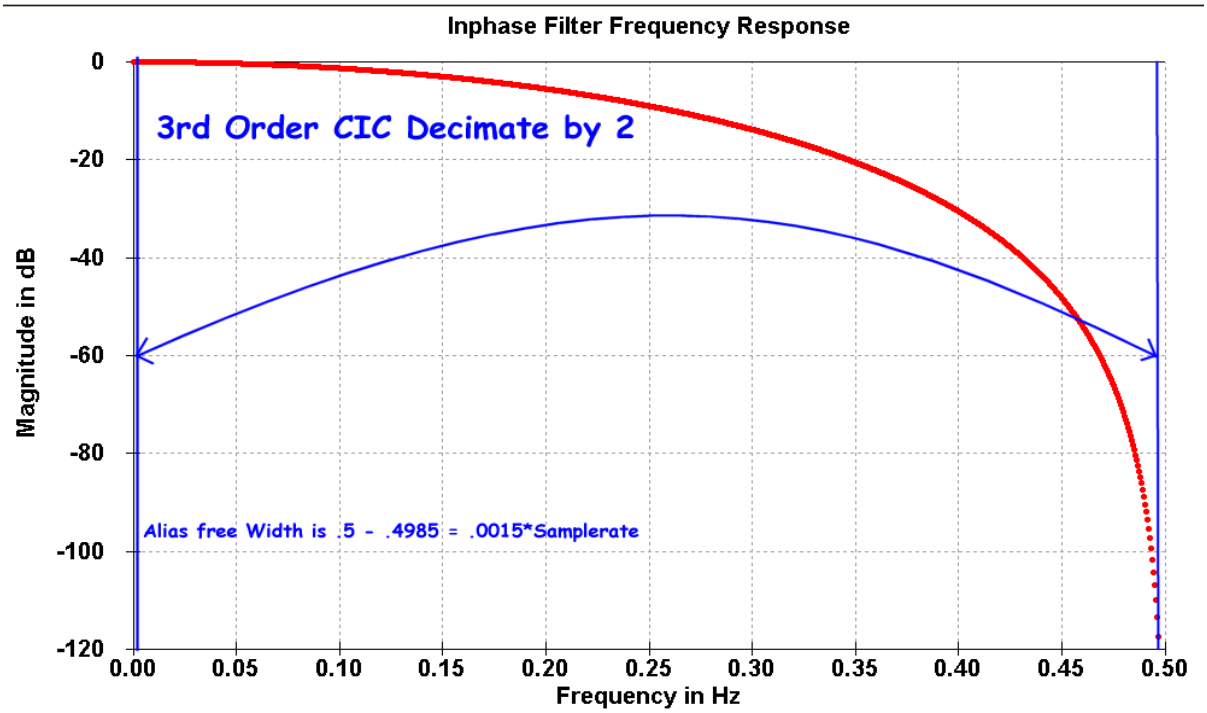




This is the implementation of the CIC order 3 complex decimation filter. Note there is a gain factor to keep total gain at unity.

```
for(i=0,j=0; i<InLength; i+=2,j++)
{      //mag gn=8
    even = pInData[i];
    odd = pInData[i+1];
    pOutData[j].re = .125*( odd.re + m_Xeven.re + 3.0*(m_Xodd.re + even.re) );
    pOutData[j].im = .125*( odd.im + m_Xeven.im + 3.0*(m_Xodd.im + even.im) );
    m_Xodd = odd;
    m_Xeven = even;
}
```

The CIC filter is only useable for the first couple of stages when going from a high sample rate to a narrow final bandwidth. Below shows the frequency response for the CIC order 3 filter and the percent sample rate of alias free bandwidth it will support.

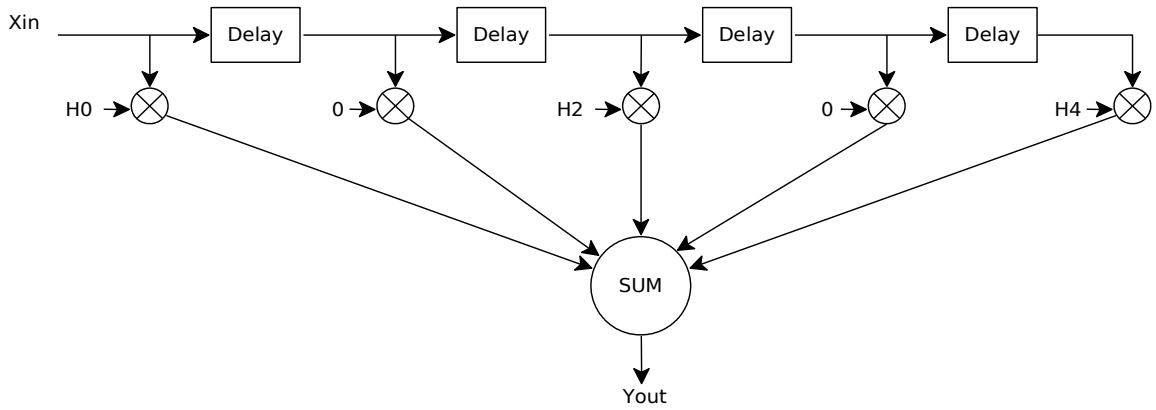
  




For Decimate by 2 stages that require a larger percentage BW of the sample rate, Half-Band filters are used.

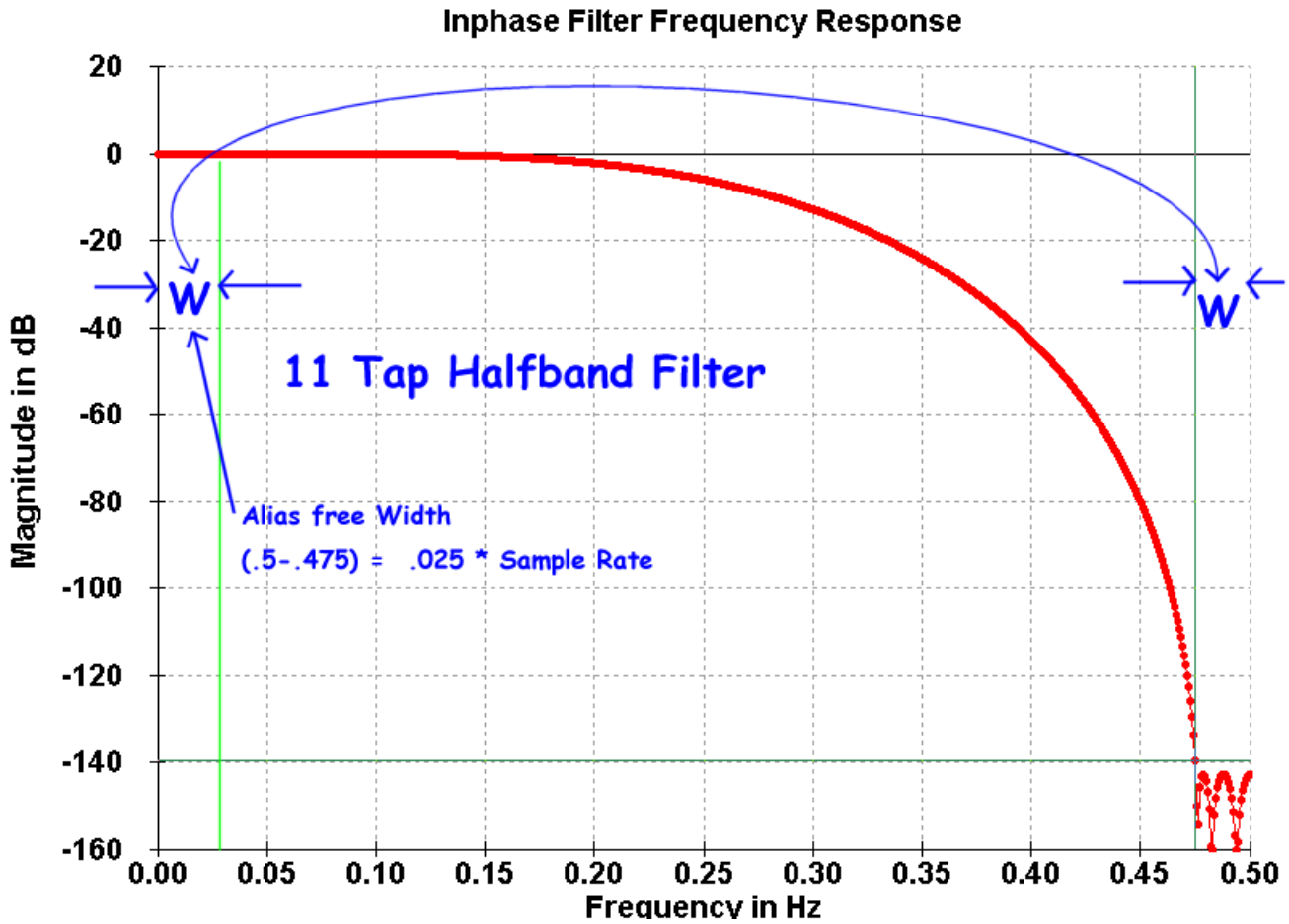
These filters are just FIR filters where every other coefficient is zero except for the center coefficient. This reduces the calculation load by almost a factor of 2 since the zero coefficients need not be calculated. The restriction of these filters is that the transition band is always centered around Fs/4. This is almost ideal for a decimate by 2 filter stage since that is where the cut off needs to be anyway.

An example 5 Tap Half-band filter. Note that H1 and H3 are zero so need not be calculated.



Below is a magnitude plot of a 11 tap Half-band filter designed to have a maximum stop band attenuation of -140dB.




Increasing the number of taps increases the alias free bandwidth but there is a point of diminishing returns.

CuteSDR implements up to a 51 Tap filter but in actuality only up to about a 27 Tap is ever required. The first several stages use the 11Tap Half-band filter so a version of it was written that "unrolled" the inner calculation loop for faster execution.

This is a snippet of the 11 Tap Half-band calculations showing the fixed MAC operations.

```
for(int i=0; i<(InLength-11-6 )/2; i++)
{
    (*pOut).re      = H0*pIn[0].re + H2*pIn[2].re + H4*pIn[4].re + H5*pIn[5].re
                                   + H6*pIn[6].re + H8*pIn[8].re + H10*pIn[10].re;
    (*pOut++).im = H0*pIn[0].im + H2*pIn[2].im + H4*pIn[4].im + H5*pIn[5].im
                                   + H6*pIn[6].im + H8*pIn[8].im + H10*pIn[10].im;
    pIn += 2;
}
```

All other Tap sizes are executed using a more generic implementation but since these larger tap sizes are used at the lower sample rates, execution time is not an issue.

The CDownConvert class implements the Decimation stages by creating a table of pointers to each of the decimation stages based on each stages bandwidth requirements. The SetDataRate() function takes the input sample rate and final maximum required Bandwidth as parameters and returns the final sample rate from the decimate by 2 stages.

```
//loop until closest output rate is found and list of pointers to decimate by 2 stages is generated
while( (f > (m_MaxBW / HB51TAP_MAX) ) && (f > MIN_OUTPUT_RATE) )
{
	if(f >= (m_MaxBW / CIC3_MAX) )		//See if can use CIC order 3
		m_pDecimatorPtrs[n++] =
			new CCicN3DecimateBy2::CCicN3DecimateBy2;
	else if(f >= (m_MaxBW / HB11TAP_MAX) )  //See if can use fixed 11 Tap Halfband
		m_pDecimatorPtrs[n++] =
			new CHalfBand11TapDecimateBy2::CHalfBand11TapDecimateBy2();
	else if(f >= (m_MaxBW / HB15TAP_MAX) )  //See if can use Halfband 15 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB15TAP_LENGTH, HB15TAP_H);
	else if(f >= (m_MaxBW / HB19TAP_MAX) )  //See if can use Halfband 19 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB19TAP_LENGTH, HB19TAP_H);
	else if(f >= (m_MaxBW / HB23TAP_MAX) )  //See if can use Halfband 23 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB23TAP_LENGTH, HB23TAP_H);
	else if(f >= (m_MaxBW / HB27TAP_MAX) )  //See if can use Halfband 27 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB27TAP_LENGTH, HB27TAP_H);
	else if(f >= (m_MaxBW / HB31TAP_MAX) )  //See if can use Halfband 31 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB31TAP_LENGTH, HB31TAP_H);
	else if(f >= (m_MaxBW / HB35TAP_MAX) )  //See if can use Halfband 35 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB35TAP_LENGTH, HB35TAP_H);
	else if(f >= (m_MaxBW / HB39TAP_MAX) )  //See if can use Halfband 39 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB39TAP_LENGTH, HB39TAP_H);
	else if(f >= (m_MaxBW / HB43TAP_MAX) )  //See if can use Halfband 43 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB43TAP_LENGTH, HB43TAP_H);
	else if(f >= (m_MaxBW / HB47TAP_MAX) )  //See if can use Halfband 47 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB47TAP_LENGTH, HB47TAP_H);
	else if(f >= (m_MaxBW / HB51TAP_MAX) )  //See if can use Halfband 51 Tap
		m_pDecimatorPtrs[n++] =
			new CHalfBandDecimateBy2::CHalfBandDecimateBy2(HB51TAP_LENGTH, HB51TAP_H);
	f /= 2.0;
}
```

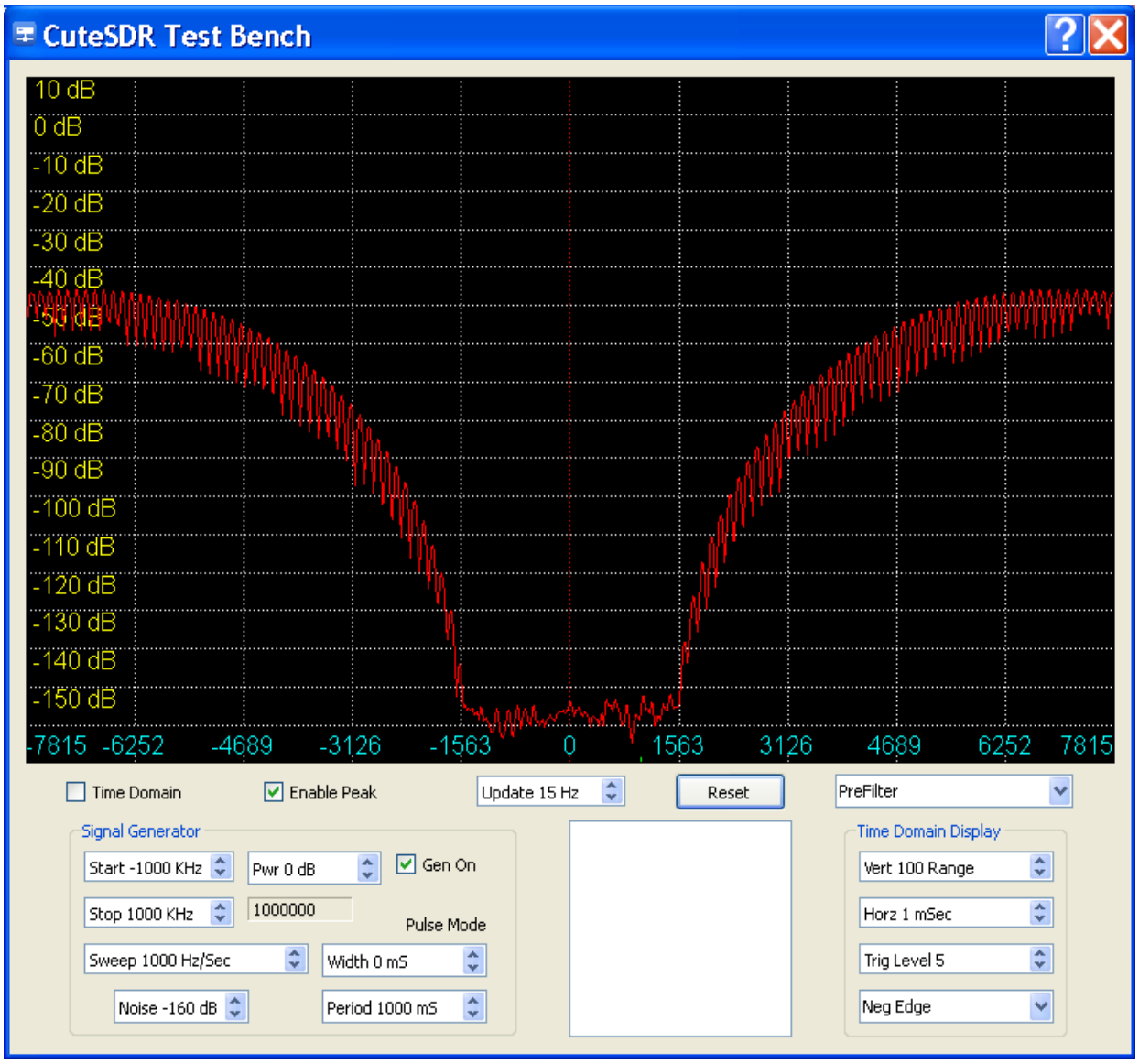
Since all the Decimate by 2 stages are derived from a pure virtual class, the list of pointers can be used to call the different stages in a simple loop. This code segment runs until a null pointer is encountered. The input buffer and output buffer is the same so each stage operates on the previous stages data.

```
int n = InLength;
int j = 0;
while(m_pDecimatorPtrs[j])
{
    n = m_pDecimatorPtrs[j++]->DecBy2(n, pInData, pInData);
}
```

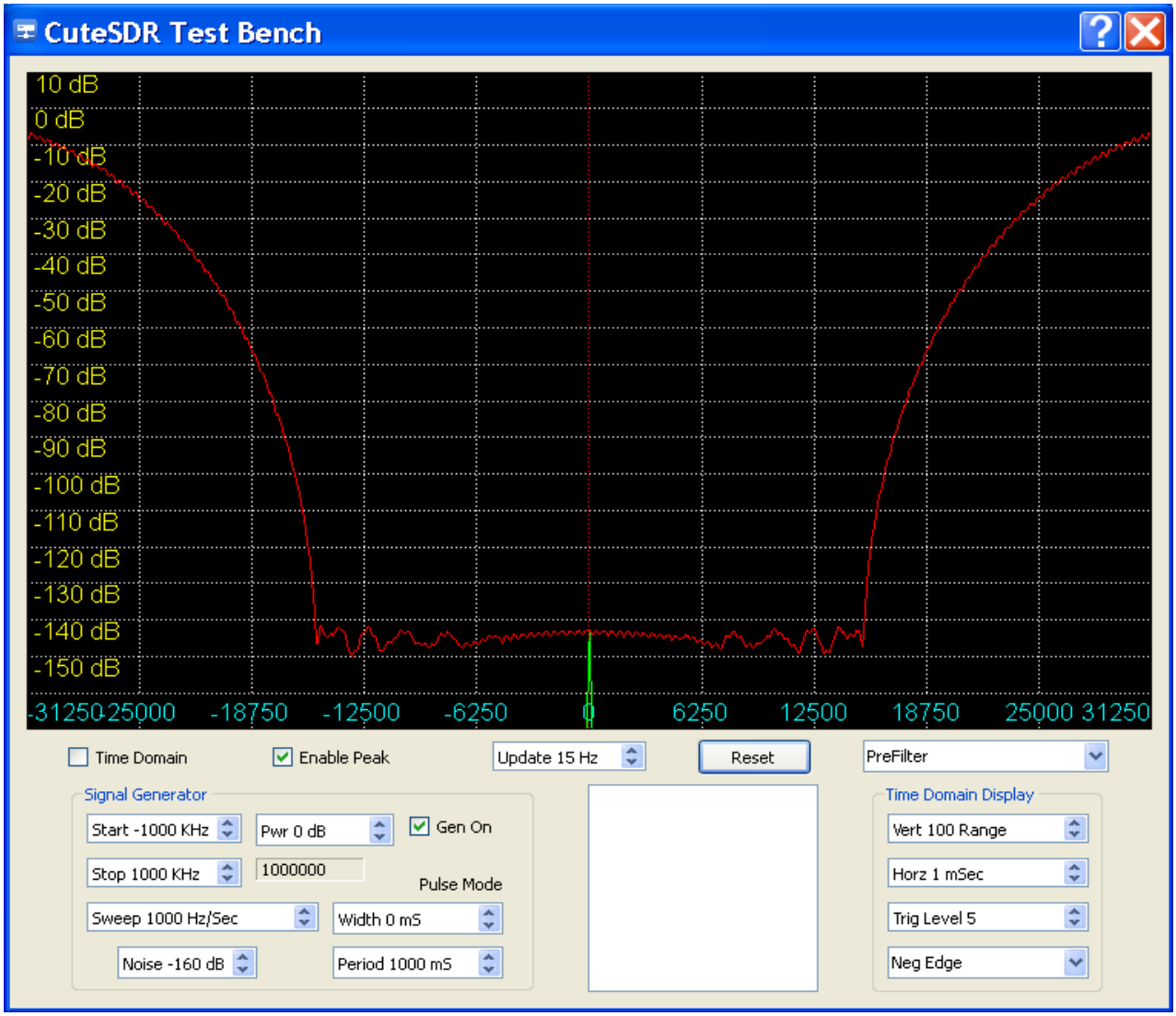
### 4.7.3 Design Verification

To test the decimation stages, a complex 0dB sweep generator was implemented that sweeps from -1MHz to 1MHz with a sample rate of 2Msps (the pass band of +-1500Hz was skipped). By selecting the narrowest bandwidth used for CW and plotting the peak spectrum of the final decimated output, the aliased power can be seen everywhere except in the required pass band. The sweep rate was 1000Hz per second so this takes about 30 minutes to complete. Note the highest signal in the pass band is <-150dB in the +-1500Hz range.



A similar sweep was done to check the widest bandwidth modes. For the +-15KHz wide bandwidth, the alias rejection is < -140dB.



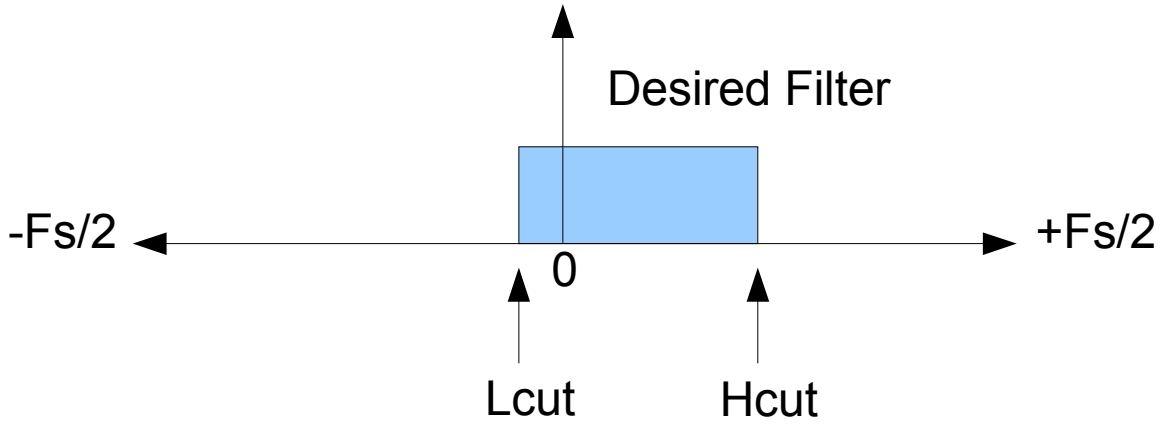

  

## *4.8. Primary Filtering (CFastFIR Class)*

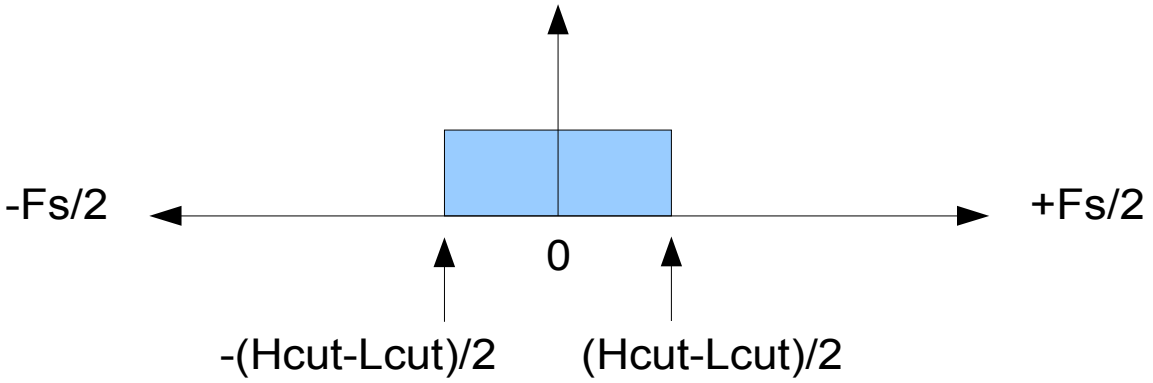
After the I/Q signal stream has been decimated to a lower sample rate, the next processing step is to filter it so that just the desired signal is allowed to pass. Since different signal types have different bandwidths, this filter must be adjustable as well as have very sharp transition regions to eliminate adjacent signal interference. The FIR filter is the filter of choice for this application as it can provide extemely sharp transition regions and very narrow bandwidths with constant phase response.

### 4.8.1 Filter Design

The filter coefficients could be pre-calculated and stored in a table if only a few filters are needed. In CuteSDR, the main filter specifications are user configurable "on the fly" so must be designed in real time. The CFastFIR class has a function SetupParameters(..) that takes as parameters the Low Cutoff and High Cutoff filter frequencies, as well as a frequency offset and the sample rate.



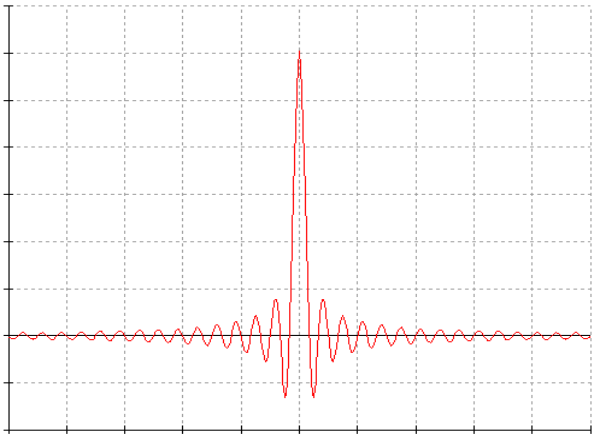

The first step in the filter design is to create a prototype Low Pass filter whose cut off frequency is half the width of the desired filter.




One way to design such a filter is by the "Windowed Sinc Filter" method. An ideal complex "brick wall" low pass FIR filter with a cut off of +-$F_c$ is described by a sinc() function.

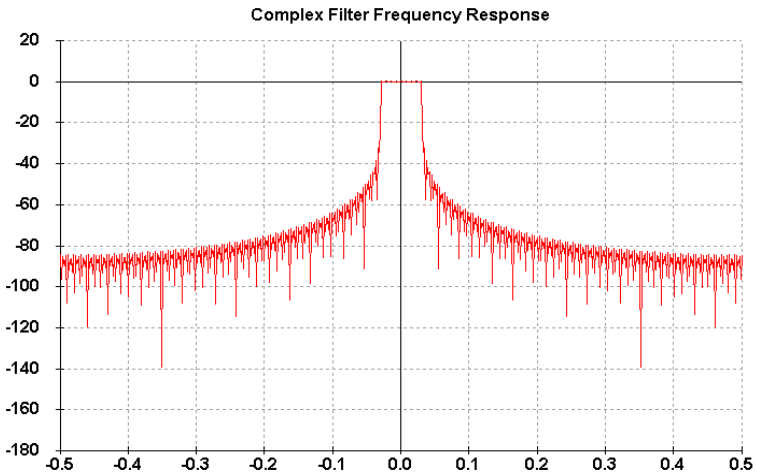
$$sinc(t) = 2F_c \frac{\sin(2\pi F_c t)}{2\pi F_c t} = \frac{\sin(2\pi F_c t)}{\pi t}$$

When t is zero, the sinc(0) term is 2 $F_c$.
This function has the following shape:

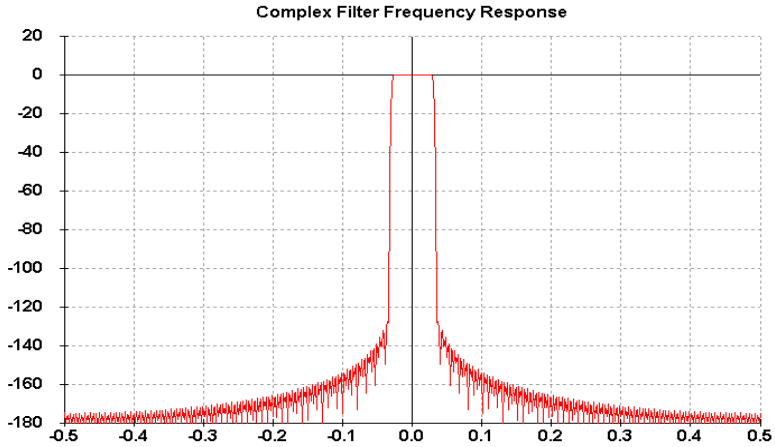


This is the impulse response of an ideal lowpass filter. In order to be an ideal filter the impulse response must be infinite which would require a FIR with an infinite number of taps and a very patient user to wait for the signal to come out the other end.

If one just truncates the filter taps, the frequency response is not very good. Below shows a 1000 tap FIR whose coefficients are just truncated:

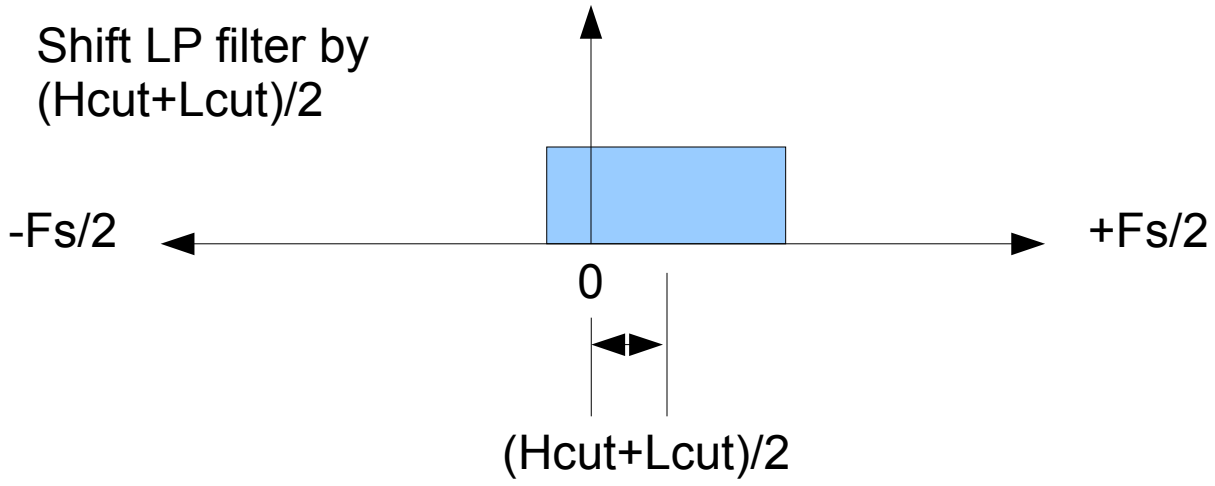



By multiplying the coefficients by a windowing function, a much better response can be obtained. Below shows the same filter as above but after applying a window function.




There are many windowing functions available and trade off transition region sharpness and stopband rejection and other characteristics. CuteSDR can be compiled to use several different windows and others can be easily added.

After a Low Pass complex filter is designed, it needs to be shifted in frequency by (HighCut+LowCut)/2 to meet the original high and low cut specification.



The complex low pass filter coefficients can be modified to shift the center frequency by multiplying them by $e^{j\omega t}$. This comes directly from the "Frequency Translation Theorem" (Google is your friend).

$$h(t)e^{(j\omega_0 t)} \leftrightarrow F(\omega-\omega_0)$$

Again using Euler's formula results in multiplying the real coefficients by $\cos(\omega_0 n)$ and the imaginary coefficients by $\sin(\omega_0 n)$. Where n is the nth index from the center coefficient.

The code segment below shows the design procedure for producing the bandpass filter coefficients:

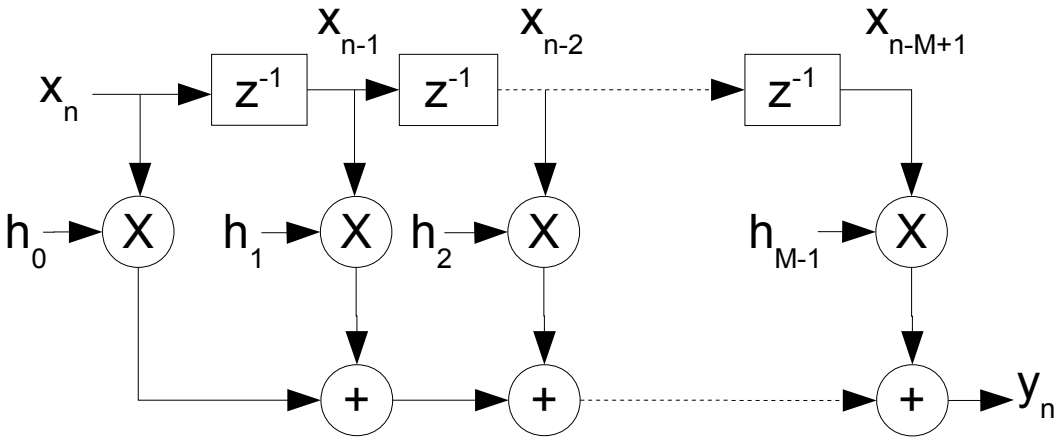
```
//create LP FIR windowed sinc, sin(x)/x complex LP filter coefficients
for(i=0; i<CONV_FIR_SIZE; i++)
{
      TYPEREAL x = (TYPEREAL)i - fCenter;
      TYPEREAL z;
      if( (TYPEREAL)i == fCenter )    //deal with odd size filter singularity where sin(0)/0==1
            z = 2.0 * nFc;
      else
            z = (TYPEREAL)sin(K_2PI*x*nFc)/(K_PI*x) * m_pWindowTbl[i];
      //shift lowpass filter coefficients in frequency by (hicut+lowcut)/2 to form bandpass filter anywhere in range
      // (also scales by 1/FFTsize since inverse FFT routine scales by FFTsize)
      m_pFilterCoef[i].re  =  z * cos(nFs * x)/(TYPEREAL)CONV_FFT_SIZE;
      m_pFilterCoef[i].im = z * sin(nFs * x)/(TYPEREAL)CONV_FFT_SIZE;
}
```

One of the input parameters to the filter generator besides the high and low cut frequencies is an offset frequency. This can be used to shift the filter by an additional amount. This is used to help in demodulating a CW signal so that it is shifted to an audible tone otherwise tuning exactly to a CW signal would just produce a DC level. By shifting the NCO in the DownConverter section also by the same amount, the CW signal will be centered in the filter and produce a tone equal to the frequency shift.

This offset could also be used to shift the output by say 12 KHz so that a low IF digital demodulator could be used such as in DRM receivers.

### 4.8.2 Filter Implementation

Implementing sharp cutoff filters require FIR filters with a very large number of coefficients which take up a large amount of CPU time when the convolution operation is implemented directly. The normal FIR structure is shown below:




In equation form this can be written as:

$$y(n)=\sum_{k=0}^{M-1} h(k)\,x(n-k)$$

or

$$y(n)=h(k)*x(n)$$

where * represents the operation of convolution. For every new sample this requires multiplying and adding all M-1 previous samples by the corresponding coefficients. As M gets large this becomes CPU intensive.

A useful property of the Fourier transform is that convolution in the time domain is the same as point by point multiplication in the Frequency domain which one gets after performing an FFT on time domain data.

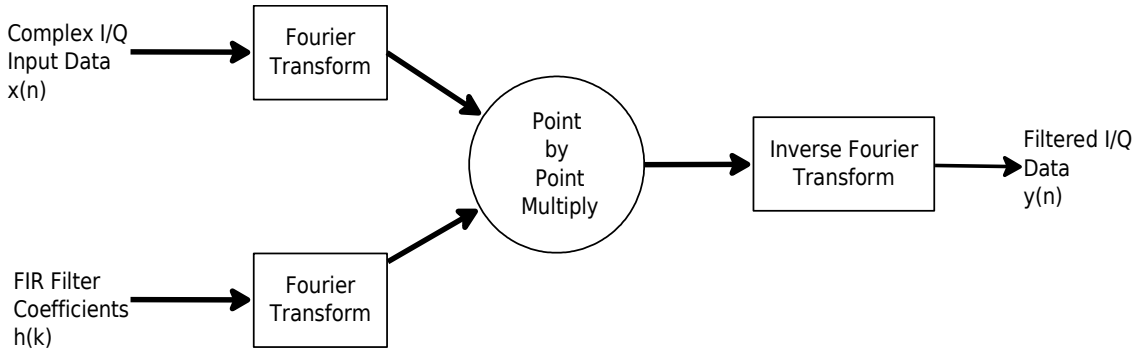
$$F\{h*x\}=F\{h\}\cdot F\{x\}$$

where * is convolution and · is point by point multiplication.

Using the Inverse Fourier transform on this gets back to the time domain after convolution.

$h*x=F^{-1}\{F\{h\}\cdot F\{x\}\}$ For proof Google "Convolution Theorem".

What this means in a practical sense is that one can take the FFT of the input data and take the FFT of the FIR filter coefficients and just multiply them together point by point. Then take the inverse FFT and the data is filtered and back in the time domain. This operation it is called Fast Convolution Filtering.



Note that the FIR Filter Coefficients and its Fourier transform need only be calculated once or anytime the filter needs to change.

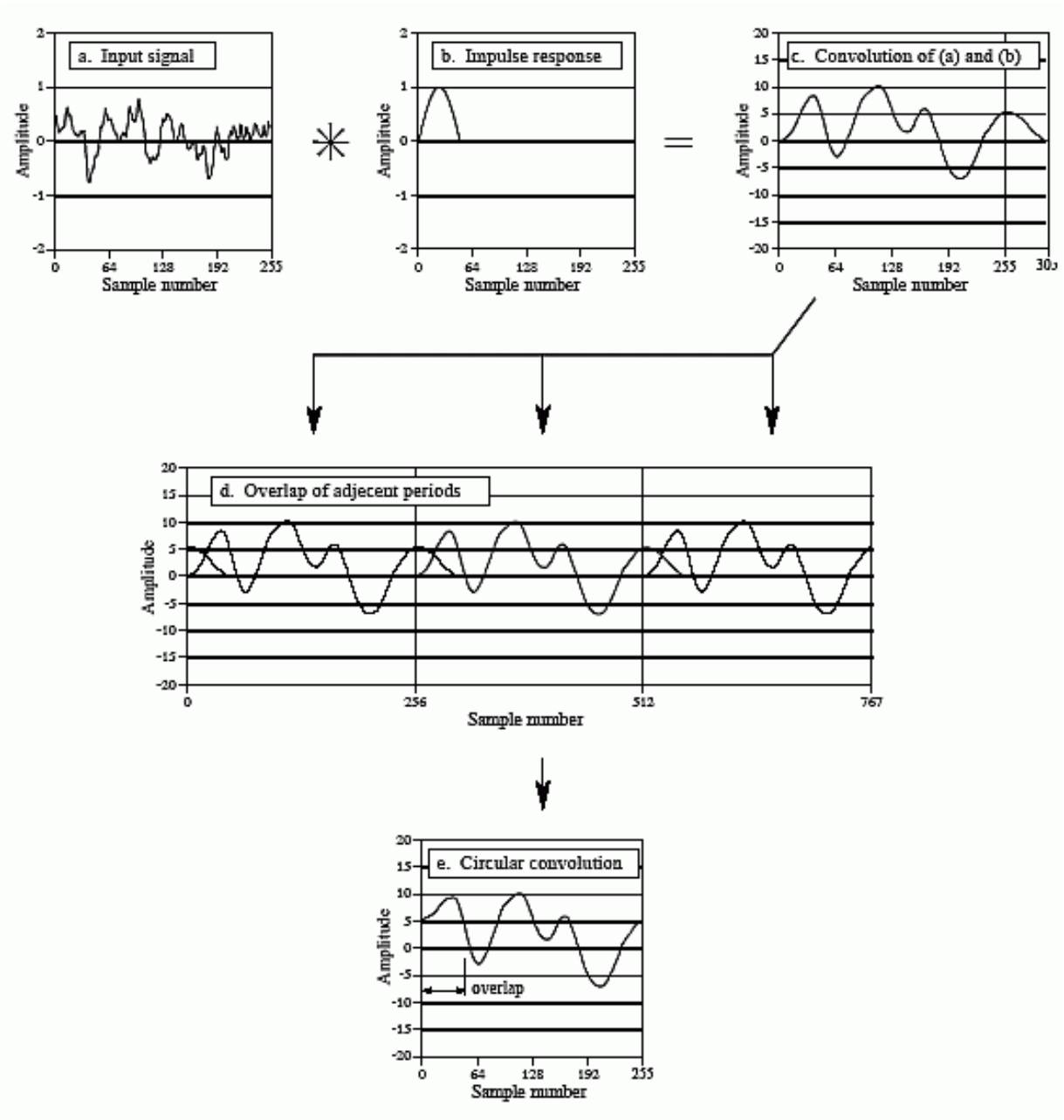
While simple in concept, the the actual implementation is not as straight forward. The problem is that the FFT method creates circular convolution which is because an N point FFT assumes that the input signal is periodic over the period of N samples. The affect of this is that some of the frequency domain data overlaps into the next FFT data corrupting part of the output data of each FFT.

The solution is to use an FFT size greater than the FIR size and zero pad the data before performing the FFT. This way the overlap occurs but we only use the data points that are not affected by it.

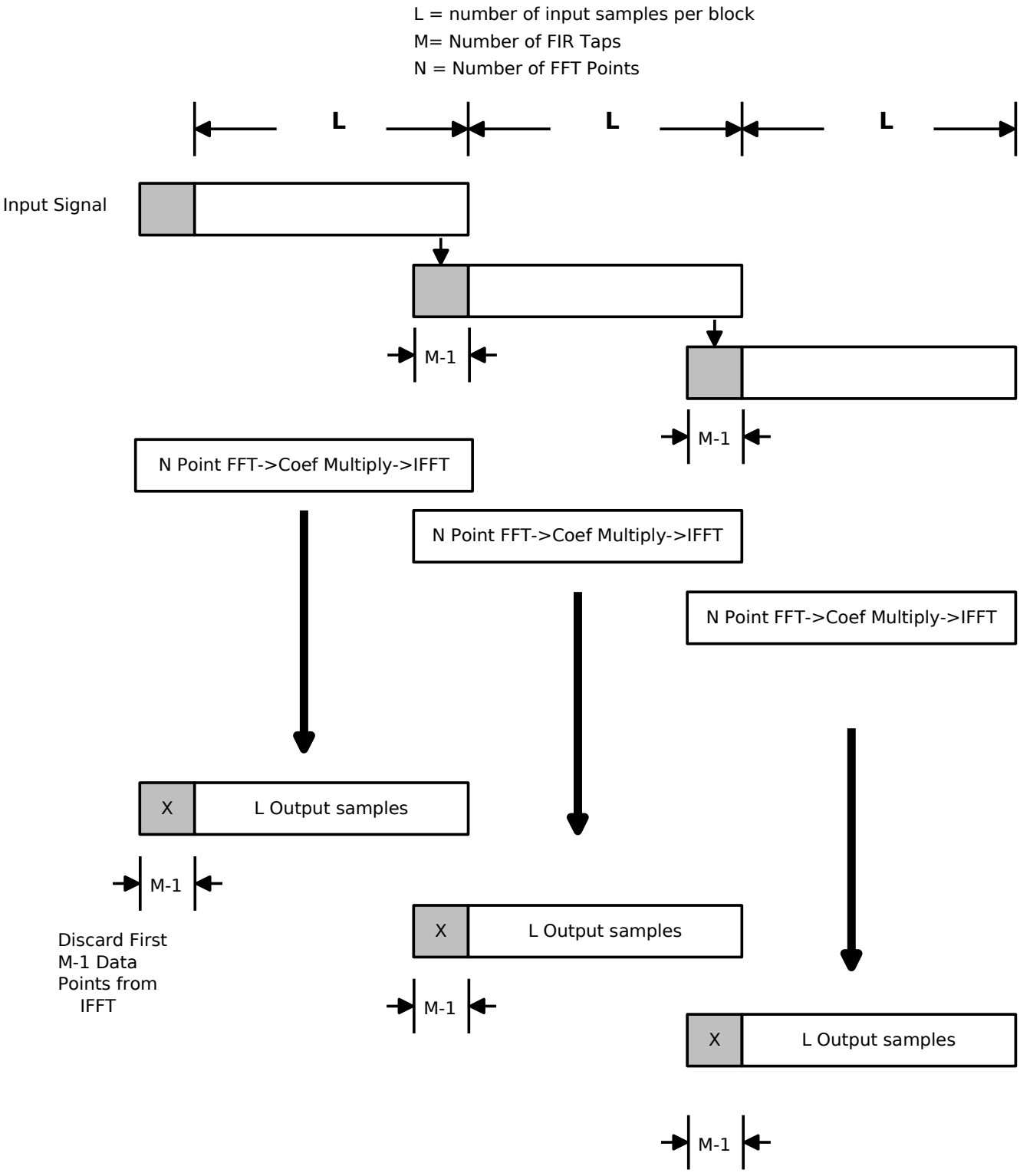
The following example from “The Scientist and Engineer's Guide to Digital Signal Processing” By Steven W. Smith, Ph.D. shows a 256 point FFT and a 51 point coefficient array convolved together using the fast convolution method.

The convolution of an N point FFT and an M point coefficient array results in a N+M-1 point output signal so the output is 256+51-1 points or 306 samples. Since the FFT output is only 256 points, these extra points wrap around into the output data array.




The way around this is to use a larger FFT and pad the data such that the FFT size is greater than the convolution output length of N+M-1 then one can just discard the overlapped section. The downsize is that part of the FFT is wasted processing zero data but it is still an efficient method of convolution. Several methods such as overlap-add and overlap-save can be used. In CuteSDR the overlap-save method was chosen as it is more straight forward. The input buffer takes L new samples and M-1 samples from the previous block. When FFT size samples are available, the FFT is performed, multiplied by the coefficients, and then an inverse FFT is performed. The first M-1 samples from the inverse FFT are discarded leaving L output samples per input block.



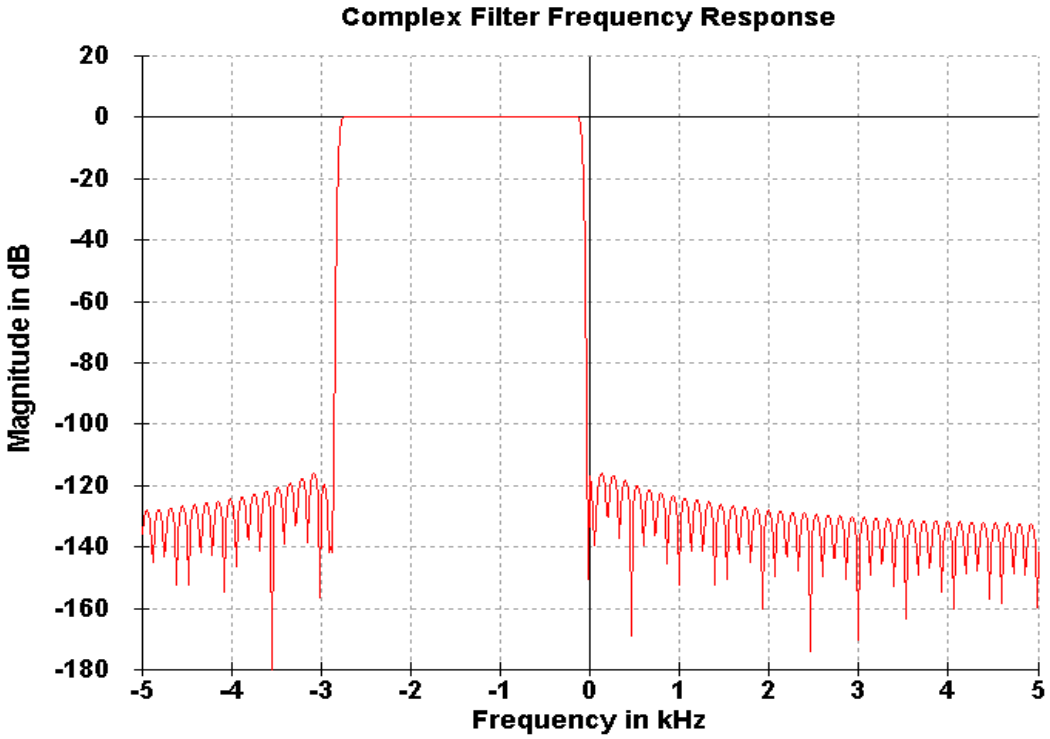
The code implementation snippet is shown below from the CFastFIR filter class. The while loop gathers enough samples to be able to perform a complete FFT on CONV_FFT_SIZE samples, perform a complex multiply by the FIR coefficients, and then perform the inverse FFT to get back to the time domain. The first CONV_FFT_SIZE-1 output samples are discarded and then the last CONV_FIR_SIZE – 1)-1 input samples are copied back to the beginning of the input buffer to get ready for the next input data block.

```
while(inlength--)
{
	j = m_InBufInPos - (CONV_FFT_SIZE - CONV_FIR_SIZE + 1) ;
	if(j >= 0 )
	{	//keep copy of last CONV_FIR_SIZE-1 samples for overlap save
		m_pFFTOverlapBuf[j] = InBuf[i];
	}
	m_pFFTBuf[m_InBufInPos++] = InBuf[i++];
	if(m_InBufInPos >= CONV_FFT_SIZE)
	{	//perform FFT -> complexMultiply by FIR coefficients -> inverse FFT on filled FFT input buffer
		m_Fft.FwdFFT(m_pFFTBuf);
		CpxMpy(CONV_FFT_SIZE, m_pFilterCoef, m_pFFTBuf, m_pFFTBuf);
		m_Fft.RevFFT(m_pFFTBuf);
		for(j=(CONV_FIR_SIZE-1); j<CONV_FFT_SIZE; j++)
		{	//copy FFT output into OutBuf minus CONV_FIR_SIZE-1 samples at beginning
			OutBuf[outpos++] = m_pFFTBuf[j];
		}
		for(j=0; j<(CONV_FIR_SIZE - 1);j++)
		{	//copy overlap buffer into start of fft input buffer
			m_pFFTBuf[j] = m_pFFTOverlapBuf[j];
		}
		//reset input position to data start position of fft input buffer
		m_InBufInPos = CONV_FIR_SIZE - 1;
	}
}
```

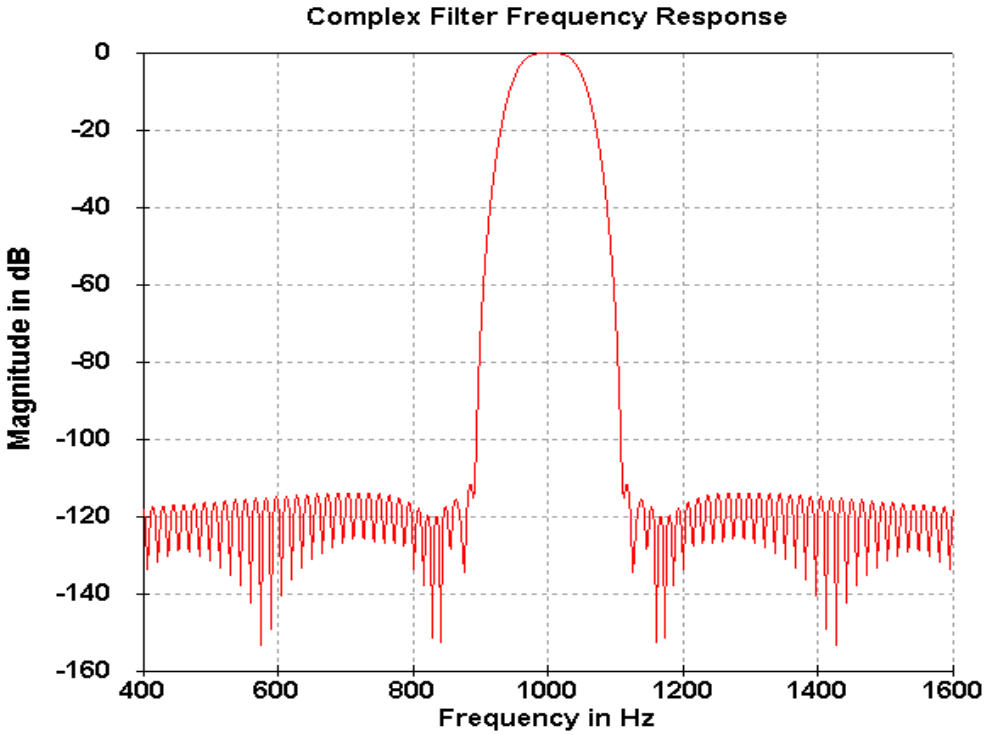
### 4.8.3 Filter Analysis

The following graphs are of a couple of CuteSDR's filters. CuteSDR uses a 1025 tap FIR design for all its filters and a 2048 point FFT to implement the fast convolution filter. It is interesting that for SSB reception, this is all one needs to do to get sideband audio. Just take the output of the filter and send it to the sound card(after AGC). This works for upper and lower sideband signals by just filtering out the opposite sideband. Just another benefit that comes from processing complex I/Q data.

This is a lower sideband filter from -2800Hz to -100Hz.



The following is a 100Hz wide CW filter with a 1000Hz tone output. For sharper and narrower filters, one could decimate down further, filter, then interpolate back up to the sound card rate.



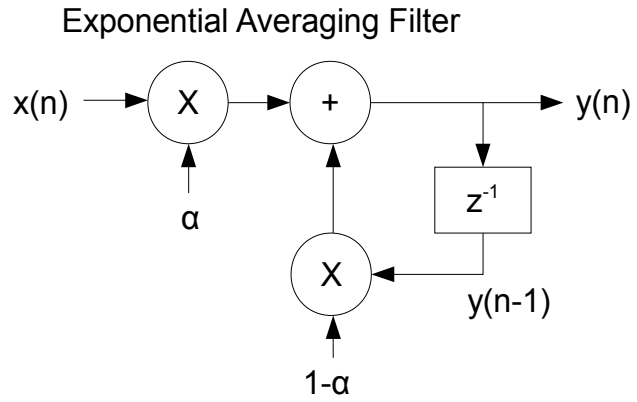
## *4.9. SMeter (CSMeter Class)*

This class takes the I/Q data after the main filter and generates power information for use in the S-Meter display. This class does not implement the actual GUI meter graphics, it just extracts the power data and performs averaging on it for the GUI display to use.

### 4.9.1 Design

The standard S-Meter is implemented using two averagers with two time constants. An attack time constant of around 10 milliSeconds is used when a signal is increasing in level and a decay time constant >= 500 milliSeconds is used when the signal is decreasing. This allows the meter to react quickly to a signal and then hold the reading long enough for the user to see it.

The simplest averaging filter to implement is the exponential moving average filter. This filter is a simple IIR filter with one delay element and is equivalent to an analog first order RC filter.




This is easily implemented with one line of code:

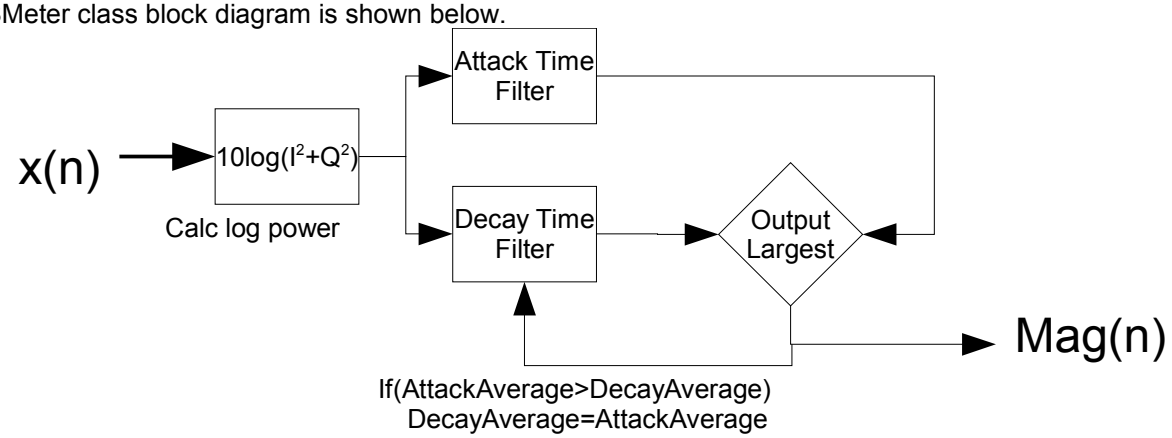
Ave = ALPHA*newsample + (1-ALPHA)*Ave;

ALPHA is related to the time constant of this filter by:

$ALPHA = 1 - e^{-T/Tau}$ where T is the sample period in seconds.

The time constant of an equivalent RC filter is Tau=1/RC.

The SMeter class block diagram is shown below.




The block calculates the log power from the input I and Q signal stream and it is averaged using the attack and Decay averagers. If the Attack average is greater than the Decay average then it is used as the output. The Decay average is also forced to the Attack average so that it "charges" up at the same attack rate. If the attack average is less than the Decay average then the slower Decay average is the output.

### 4.9.2 Implementation

The code segment that implements this is shown here:

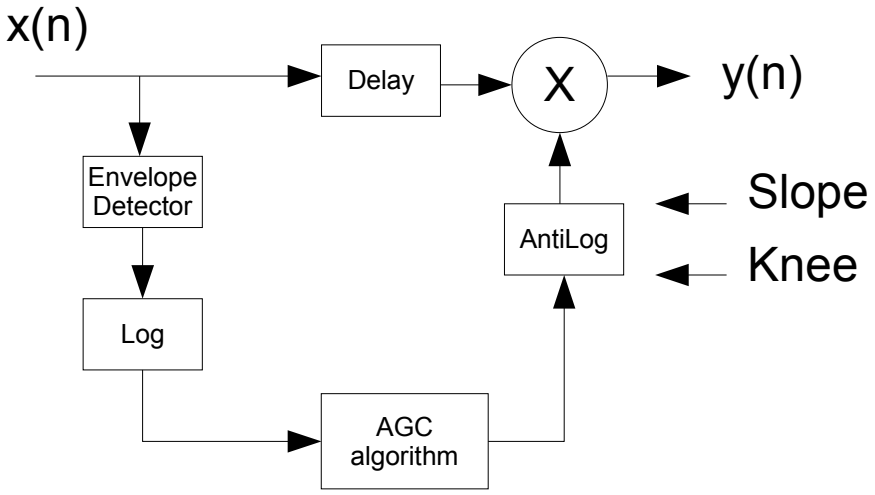
```
//convert I/Q magnitude to dB power
TYPEREAL mag = 10.0*log10((in.re*in.re+in.im*in.im)/ MAX_PWR + 1e-50);
//calculate attack and decay average
m_AttackAve = (1.0-m_AttackAlpha)*m_AttackAve + m_AttackAlpha*mag;
m_DecayAve = (1.0-m_DecayAlpha)*m_DecayAve + m_DecayAlpha*mag;
if(m_AttackAve>m_DecayAve)
{       //if attack average is greater then must be increasing signal
        m_AverageMag = m_AttackAve;     //use attack average value
        m_DecayAve = m_AttackAve;       //force decay average to attack average
}
else
{       //is decreasing strength so use decay average
        m_AverageMag = m_DecayAve;      //use decay average value
}
```

### *4.10. AGC (CAgc Class)*

Most radio signals vary in strength over a wide dynamic range due to variable path distances, propagation changes, power differences, etc. This signal variation can be over 100dB and is much greater than the human ear dynamic range. This requires some form of gain control to be inserted in the signal chain after the main bandpass filter.

In analog receivers, automatic gain control(AGC) is usually distributed across several gain stages to keep the individual RF and IF amplifiers from saturating. Since CuteSDR uses double precision floating point for all its signal processing, it can easily handle the signal dynamic range and the AGC function can operate at a single point in the signal chain. A reasonable point is right after the main bandpass filtering ahead of the demodulation stage. Most AGC algorithms operate in the log domain instead of the linear domain due to the large dynamic range of the signals.

Below is a block diagram of the basic AGC system. It can be analyzed in two stages. First is the static or DC gain of the AGC and the user inputs that affect the gain. Second is the dynamic characteristics of the AGC which deal with time varying issues. The Delay line in the system allows the AGC to operate on the signal “in the future” to begin AGC operation before it gets to the output.



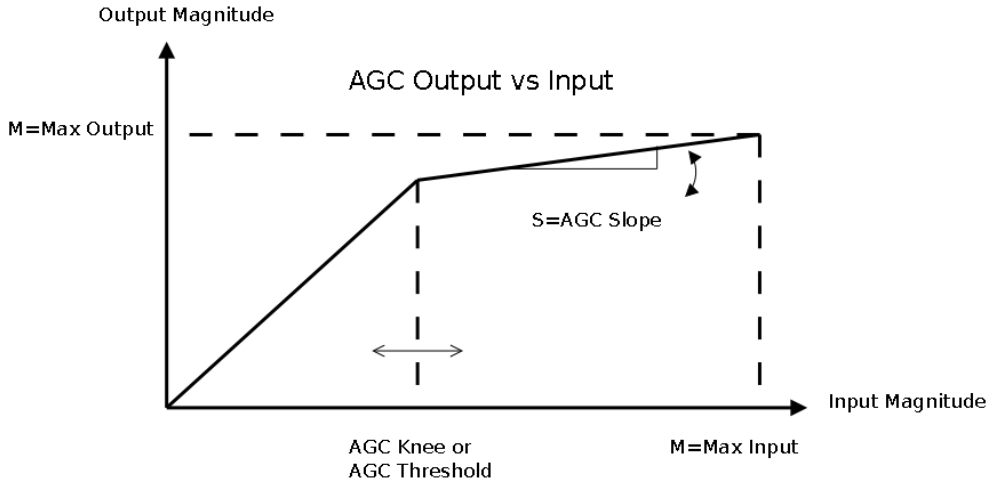

  

### 4.10.1 Static AGC Gain Design

Below is a plot of a typical AGC gain transfer function where there are two user adjustable parameters of Threshold and Slope. Signals below the threshold level are multiplied by a fixed gain and above the threshold, the AGC has a gain that is a function of the input level. The slope parameter adjusts how much gain adjustment above the threshold is used.

The Threshold adjustment is useful in cases where all the desired signals are well above the noise level so the threshold can be adjusted such that the background noise is reduced like a soft squelch control.

The Slope adjustment is a more subtle control and makes stronger signals sound louder than weaker ones which can be more pleasing to the ear. A slope of zero would make all the signals above the threshold appear as the same level which is probably preferred for digital modes or FM.

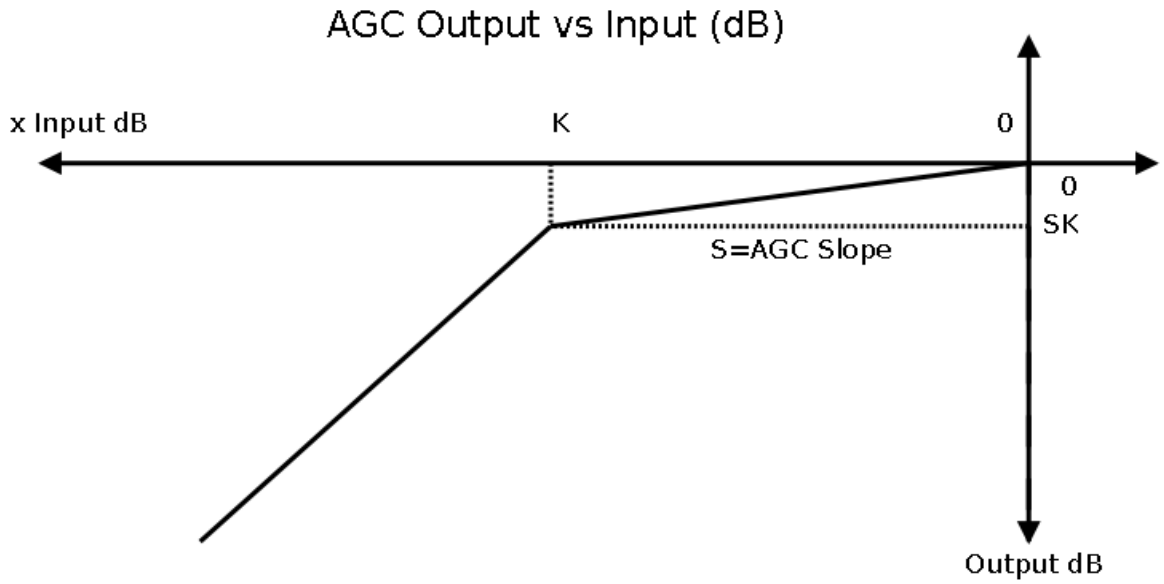



For simplicity, let M equal the maximum input and the maximum output level. Let S equal the slope of the function above the threshold.

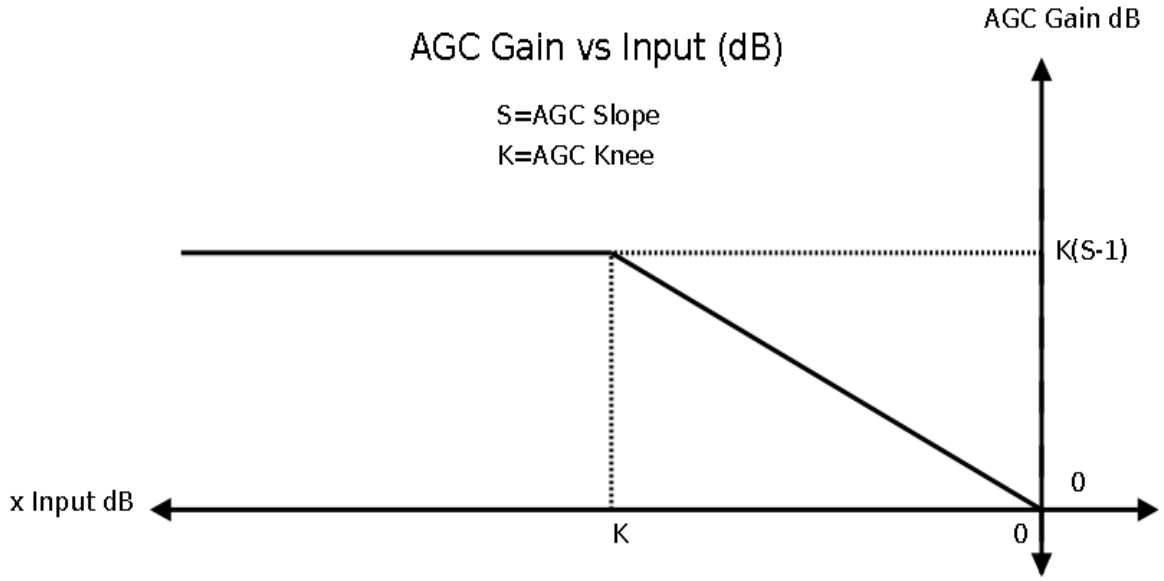
To implement this AGC function we need to derive an AGC gain vs input magnitude function. Since the gain slope is in dB, one needs to create a new function of the above Output vs Input function where the input and output are in dB. Another way to look at it is, what function is needed to multiply the input by in order to achieve the above output function. Due to the discontinuity at the threshold, one can break the transfer equation in to two parts. Below the threshold one just multiplies the input by a constant and above the threshold the gain is a function of the input magnitude.

Let the maximum input and output magnitudes equal 0dB for simplicity and the AGC gain function in dB looks like this:




Next create the AGC Gain vs Input magnitude in dB.




Below the Threshold Knee K: $Gain_{dB}(x) = K(S-1)$

Above the Threshold: $Gain_{dB}(x) = x(S-1)$

Converting back to the linear domain to be able to multiply by the input signal gives:

Below the Threshold Knee K: $Gain(x) = 10^{K(S-1)}$

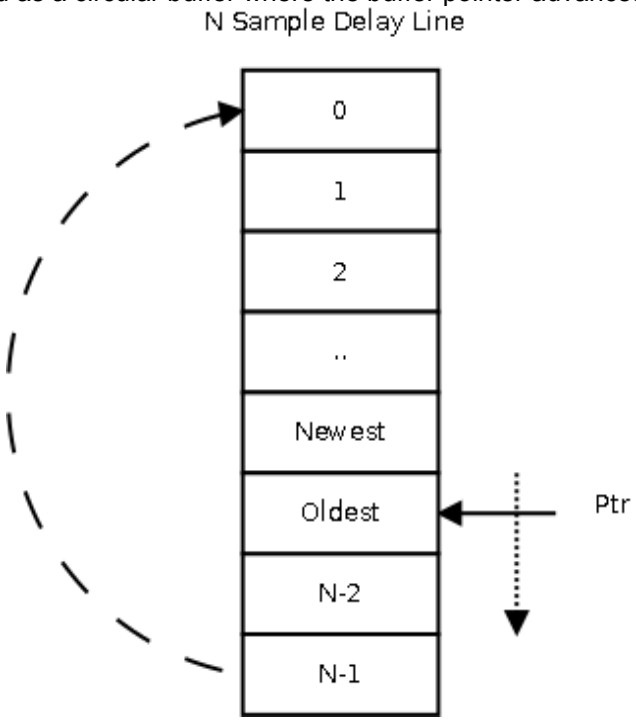
Above the Threshold: $Gain(x) = 10^{magdB(x)(S-1)}$ where magdB(x) is the dB magnitude of the input after being processed by the AGC algorithm.

### 4.10.2 Dynamic AGC Design

For signals whose power is relatively constant like FM, most digital modes, and even AM, the AGC algorithm can be relatively simple and need only adjust the gain based on a slow average of the signal to adjust for propagation changes. For signals where the information is contained in the amplitude of the signal such as SSB, AGC operation will by definition distort the signal so a compromise must be achieved that will allow the signal's amplitude information to pass undistorted and yet be able to compensate the gain for overall signal strength variations.

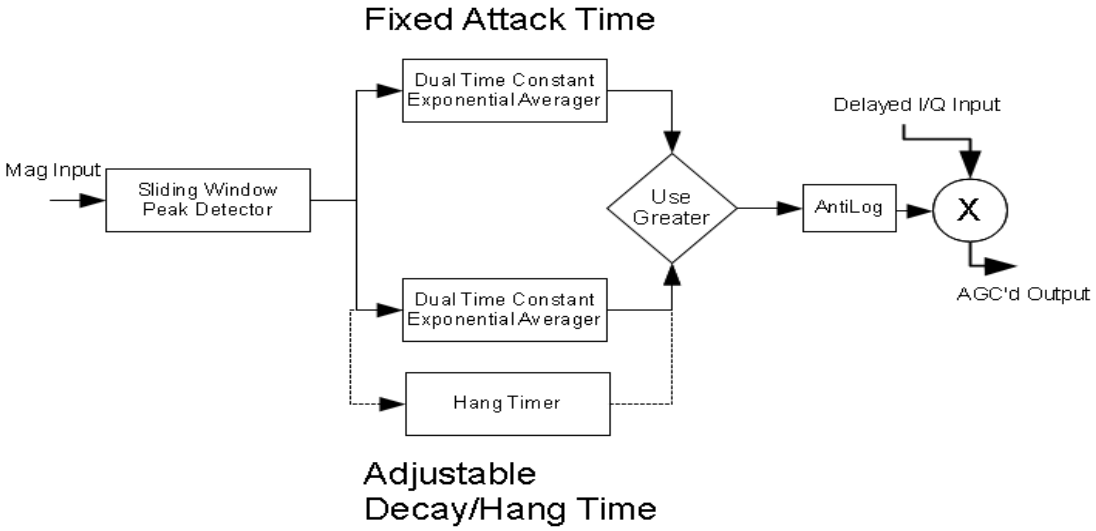
The complex input samples are delayed in order for the AGC to begin acting on the samples a little ahead of time. The delay line is implemented as a circular buffer where the buffer pointer advances instead of shifting the data.




The code segment for the delay line is:

```
//Get delayed sample of input signal
delayedin = m_SigDelayBuf[m_SigDelayPtr];
//put new input sample into signal delay buffer
m_SigDelayBuf[m_SigDelayPtr++] = in;
if( m_SigDelayPtr >= m_DelaySamples)    //deal with delay buffer wrap around
    m_SigDelayPtr = 0;
```

The design of an AGC system requires a lot of experimentation and trying different schemes. The following diagram is what was finally settled on:




The first step is to obtain the magnitude of the input I/Q signals. Two ways were implemented with no significant difference between them.

$$mag = \log\left(\sqrt{(I^2+Q^2)}\right)$$

or

$$mag = \log\left(max\left(|I|,|Q|\right)\right)$$

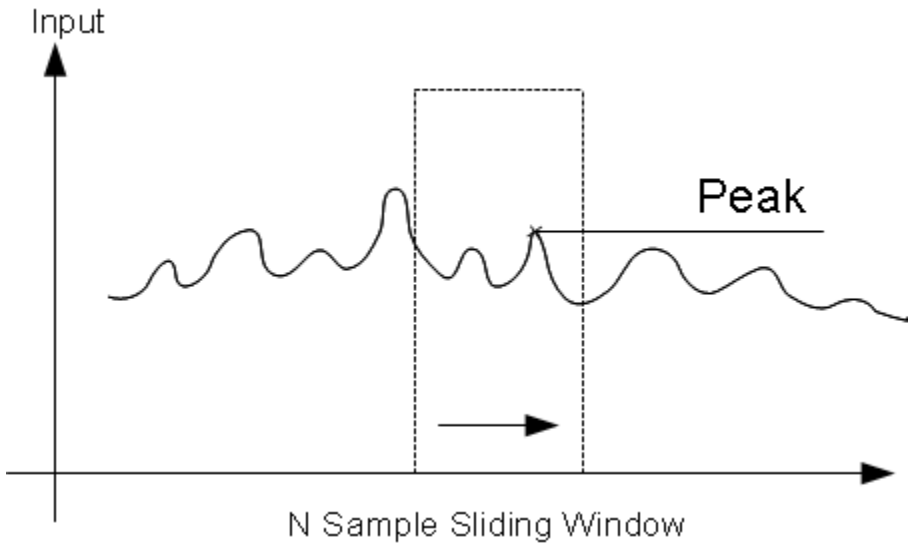
Scaling values are applied so that the maximum is 0dB and also a minimum dB value is added to keep the log function from blowing up if I and Q are zero.

$$mag = \log\left(\frac{\sqrt{(I^2+Q^2)}}{32767}+MINVAL\right)$$

The final range of mag is from -8.0 to 0.0 corresponding to -160 to 0 dB of actual signal magnitude. This is the instantaneous per sample magnitude.

After deriving the instantaneous magnitude, a peak detector is implemented. A sliding window peak detector is used that outputs the peak magnitude over a specified number of samples. This is like a running average filter except it calculates the peak instead of the average value. A few optimizations can be used to reduce the CPU load by only searching through the last N samples when necessary. If the new sample going into the sample buffer is greater than the current peak or if the last sample being removed from the buffer is not equal to the current peak, then there is no need to search through the buffer again.

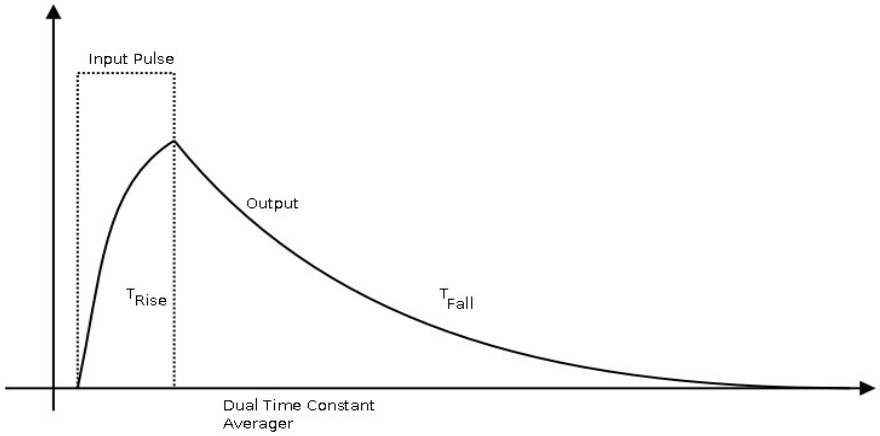
  




```
 //create a sliding window of 'm_WindowSamples' magnitudes and output the peak value within the sliding window
TYPEREAL tmp = m_MagBuf[m_MagBufPos];   //get oldest mag sample from buffer into tmp
m_MagBuf[m_MagBufPos++] = mag;                  //put latest mag sample in buffer;
if( m_MagBufPos >= m_WindowSamples)             //deal with magnitude buffer wrap around
    m_MagBufPos = 0;
if(mag > m_Peak)
{
    m_Peak = mag;   //if new sample is larger than current peak then use it, no need to look at buffer values
}
else
{
    if(tmp == m_Peak)               //tmp is oldest sample pulled out of buffer
    {           //if oldest sample pulled out was last peak then need to find next highest peak in buffer
                m_Peak = -8.0;          //set to lowest possible value to find next max peak
                //search all buffer for maximum value and set as new peak
                for(int i=0; i<m_WindowSamples; i++)
                {
                        tmp = m_MagBuf[i];
                        if(tmp > m_Peak)
                                m_Peak = tmp;
                }
    }
}
```

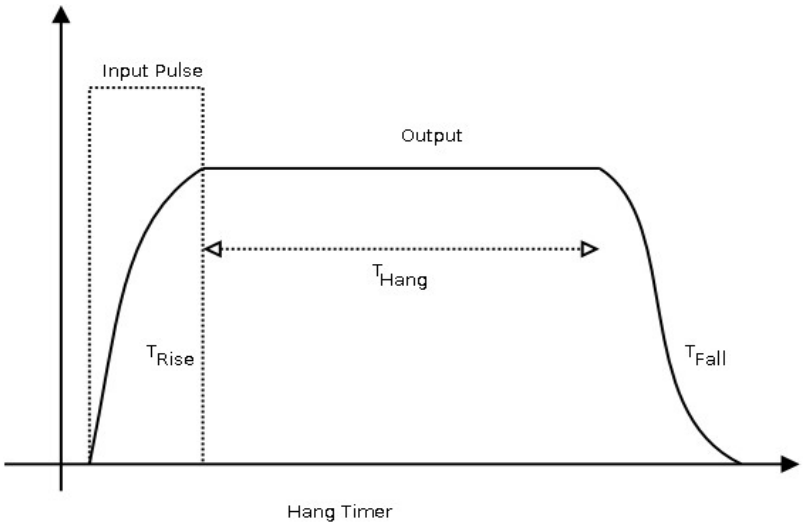
The Peak detector output is then averaged by two separate exponential averagers, The fast attack one is fast to quickly react to a fast rising signal, and an adjustable decay averager that maintains the gain at a much slower rate so as not to distort the signal.




The dual time constant averager code looks at each new sample and if it is greater than the current average it assumes a rising waveform and uses the rising time constant. If it is less than than the current average it assumes a falling waveform and uses the falling time constant.

```
if(m_Peak>m_AttackAve)  //if magnitude is rising (use m_AttackRiseAlpha time constant)
    m_AttackAve = (1.0-m_AttackRiseAlpha)*m_AttackAve + m_AttackRiseAlpha*m_Peak;
else                    //else magnitude is falling (use  m_AttackFallAlpha time constant)
    m_AttackAve = (1.0-m_AttackFallAlpha)*m_AttackAve + m_AttackFallAlpha*m_Peak;
```

The decay time can also be selected to be a “Hang” timer where the gain is held constant for a specified time then quickly decays.



The code is modified such that the Decay fall time is controlled by a counter that holds the current average as long as the hang time has not expired and the magnitude is still falling.

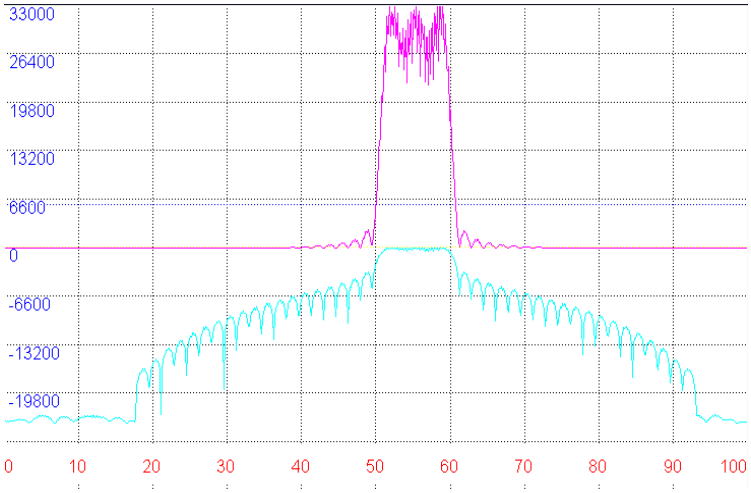
```
if(m_Peak>m_AttackAve)  //if power is rising (use m_AttackRiseAlpha time constant)
    m_AttackAve = (1.0-m_AttackRiseAlpha)*m_AttackAve + m_AttackRiseAlpha*m_Peak;
else                            //else magnitude is falling (use  m_AttackFallAlpha time constant)
    m_AttackAve = (1.0-m_AttackFallAlpha)*m_AttackAve + m_AttackFallAlpha*m_Peak;

if(m_Peak>m_DecayAve)   //if magnitude is rising (use m_DecayRiseAlpha time constant)
{
    m_DecayAve = (1.0-m_DecayRiseAlpha)*m_DecayAve + m_DecayRiseAlpha*m_Peak;
    m_HangTimer = 0;        //reset hang timer
}
else
{       //here if decreasing signal
    if(m_HangTimer<m_HangTime)
            m_HangTimer++;  //just inc and hold current m_DecayAve
    else    //else decay with m_DecayFallAlpha which is RELEASE_TIMECONST
            m_DecayAve = (1.0-m_DecayFallAlpha)*m_DecayAve + m_DecayFallAlpha*m_Peak;
}
```

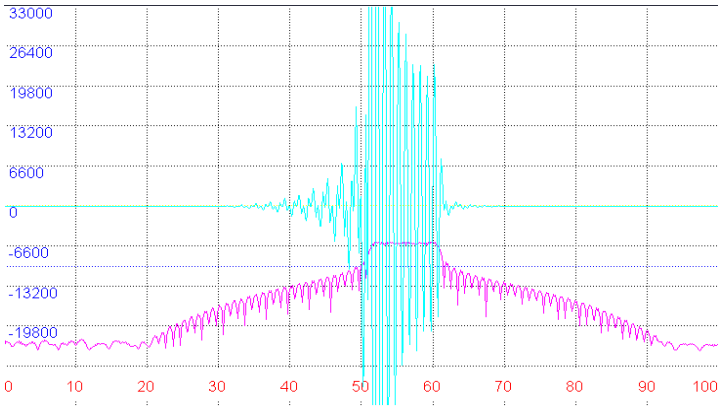
### 4.10.3 AGC Testing

The built in TestBench was used to examine various signals in the AGC algorithm while applying different signals.

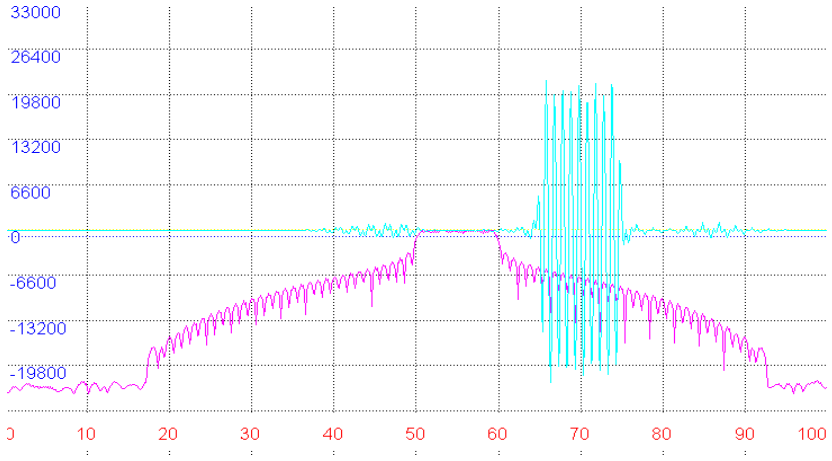
Below is a plot of a rectangular 10mSec I/Q sin/cos input pulse that goes from -140dB to 0dB. The top trace is the linear magnitude going from 0 to 32768. The bottom trace is the log magnitude multiplied by 3000 to be similar in size.



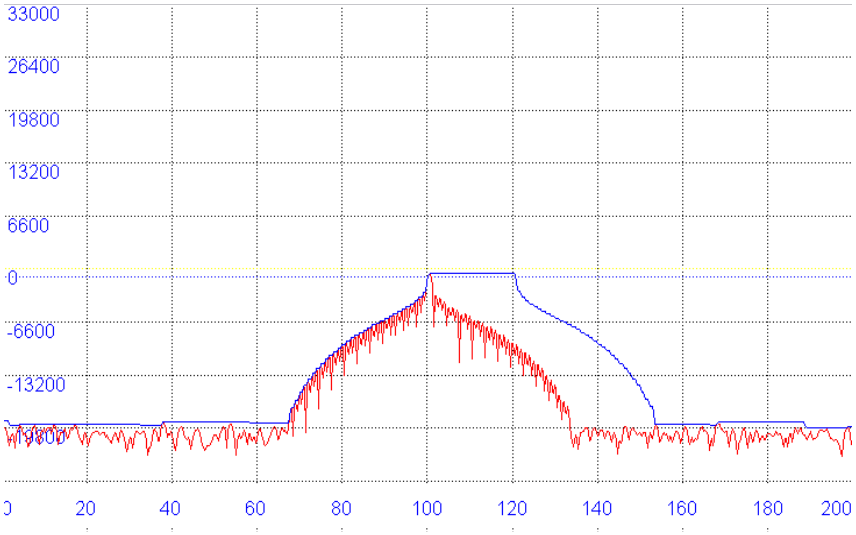

Note the ripple response before and after the pulse. This is due to the impulse response of the main 1025 tap FIR filter and is very pronounced when viewed in the log domain. This affect requires that the input signal be delayed and the peak detector to be long enough to cause the AGC to act well before the pulse otherwise the impulse response is audible as a buzz on the signal just before and after an RF pulse. The upper trace below is the signal out of the AGC block without using any input signal delay. The lower trace is the log input magnitude. Note the ripple preceding the wavefront and also the leading edge distortion due to the exponential averagers rise time constant acting on the input.



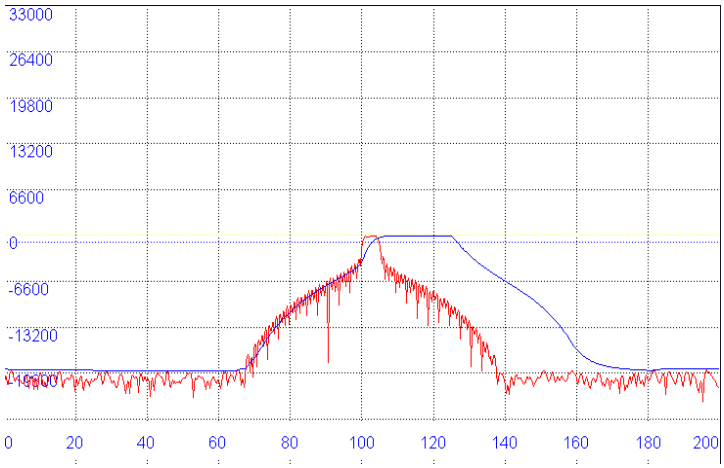
Below is the same pulse using a delayed input. There is still some ripple but this is a trade off of too much delay and also distortion of voice waveforms. In the real world, such rectangular pulses going from 0 to maximum amplitude do not exist so the impulse ringing is not such a big issue but is useful to tweak design parameters.



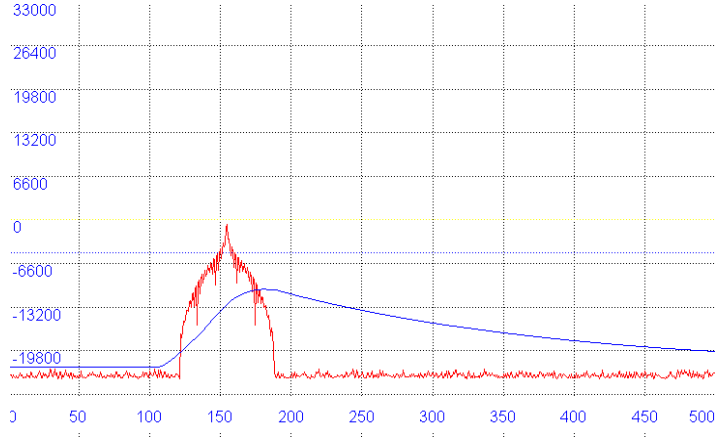

The following is a plot of the peak detector output and the log input magnitude of an input pulse. Note there are two important aspects of the peak detector. It stretches the input pulse by the window length. It also smooths out the fast changes in the noise. This is the signal that is then applied to the attack and decay averagers.



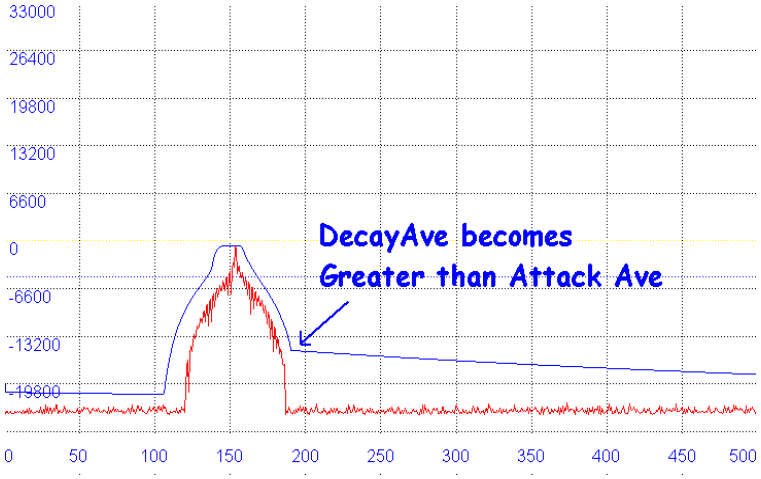
The output of the Attack Averager is shown below.



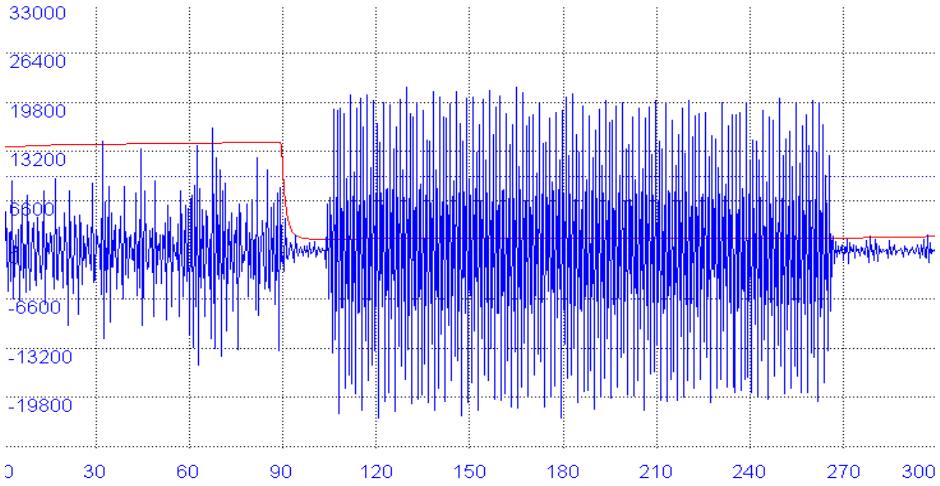

The output of the Decay Averager is shown below. Note that with the narrow pulse applied, it never reaches maximum value.



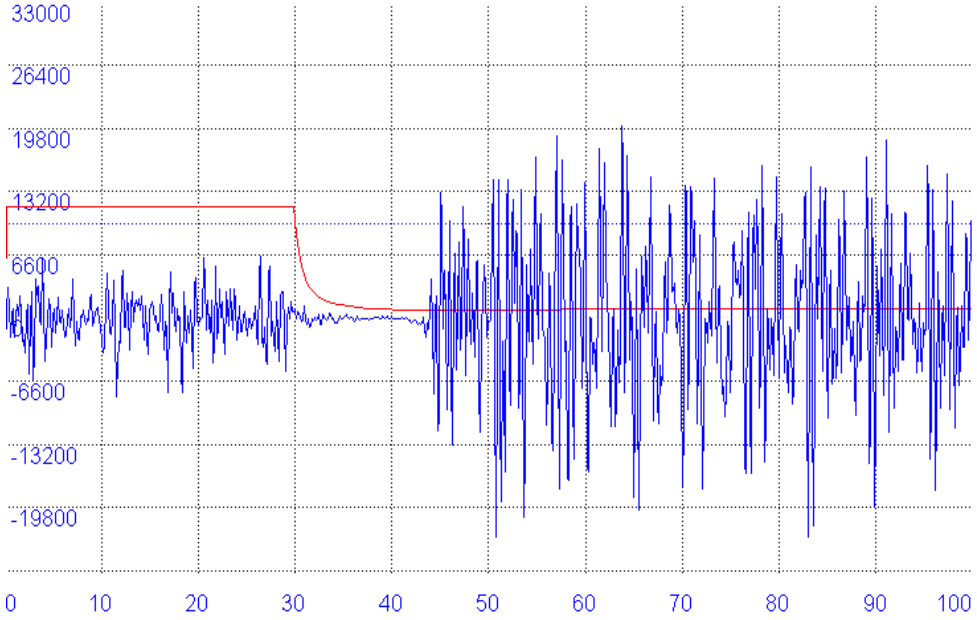

The composite output of the agc is the greater of the Attack and the Decay Averager as shown below. Note that with the narrow pulse applied, it uses the attack averager during the pulse event then switches over to the longer decay average after the event.



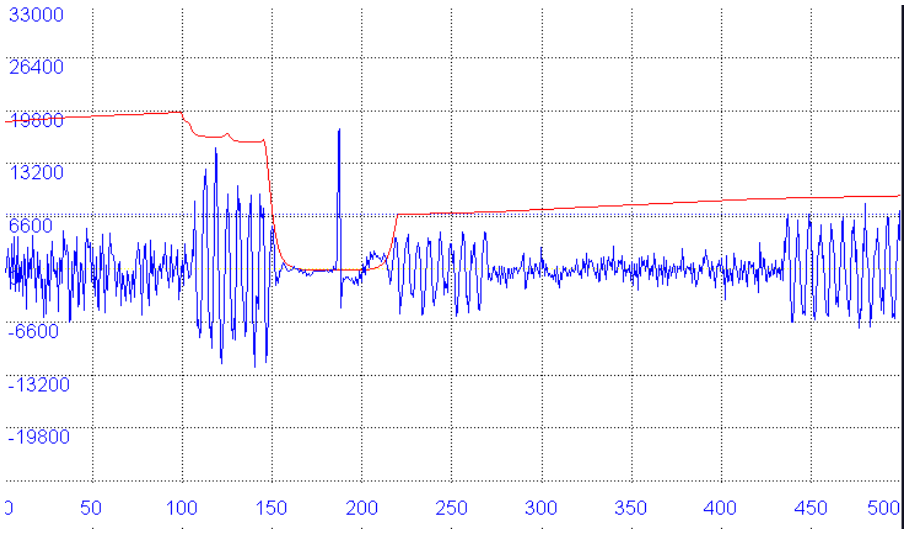
The following is the AGC gain in red acting on the first 'Dash' of a CW signal off the air. It can be seen reducing the gain a little before the CW wavefront arrives.




The following is the AGC gain in red acting on the first syllable of a SSB signal off the air. It can be seen reducing the gain a little before the voice wavefront arrives and not distorting the first part of the voice since the gain is stabilized.



The following is the AGC gain in red acting on a CW signal where a strong noise pulse occurs in the middle of the CW dash. It can be seen reducing the gain to limit the noise pulse then recover relatively quickly to get the remainder of the dash then slowly increase before the next CW character starts.



### *4.11. SSB Demodulation (CSsbDemod Class)*

This class is called when the SSB or CW mode is chosen. All it does is copy the input data to the output buffer since SSB is really not a form of modulation just a shift in frequency from audio to RF frequency. Since the preceding stages have already shifted the RF data back to baseband audio, there is nothing more to do but send it to the sound card.

There are two processing routines, one for mono audio and another for stereo. The mono just takes the real part of the I/Q data while the stereo version copies the I data to the left channel and the Q data to the right channel. This may be of some use as it creates spacial affects due to the phase changes of a signal as propagation paths change.

## *4.12. AM Demodulation (CAmDemod Class)*

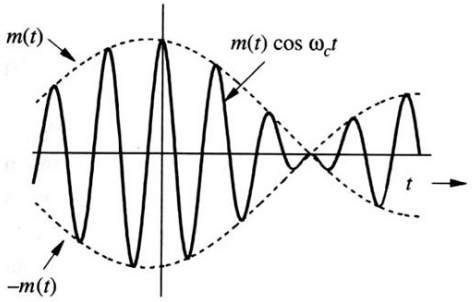
This class performs envelope AM demodulation on the I/Q data and copies the result into the output buffer.

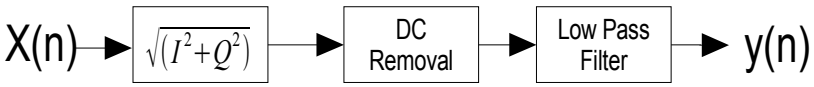
### 4.12.1 Design

An AM signal consists of a constant carrier frequency whose magnitude is the desired modulated signal. The AM signal equation is as follows:

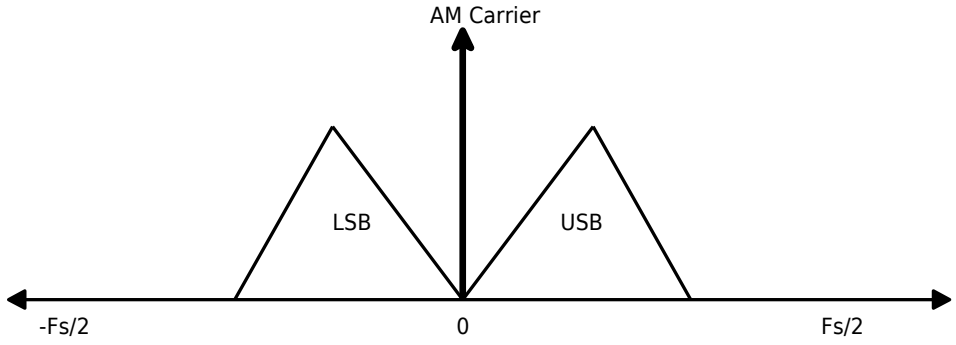
$$e(t) = A_c[1 + mx(t)]\cos(\omega_c t + \theta_c)$$

where
$A_c$ is the carrier amplitude
m is the modulation index(or modulation percentage)
x(t) is the information signal
$\omega_c$ is the carrier frequency in radians/sec
$\theta_c$ is the carrier phase




The envelope method of AM demodulation is the common method used in the analog world and can be accomplished with a diode and RC filter. We have a digitized I/Q baseband signal to work with, so it can be done by calculating the magnitude of the I/Q signal and removing the carrier component which is nearly at DC since we have shifted it to zero in the tuning to baseband stage.



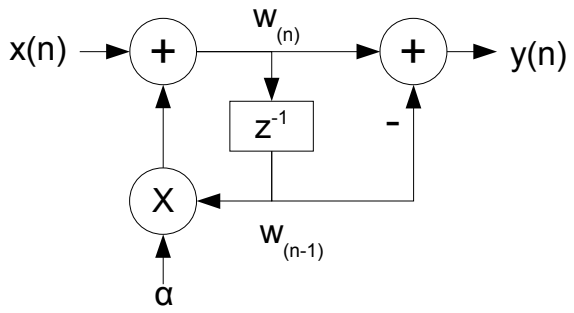

The envelope of the I/Q signal is $\sqrt{I^2+Q^2}$ . This also contains the large carrier DC component so it must be high pass filtered to remove the DC.

One method is to use a high pass filter with the following transfer function:

$$H(z)=\frac{(1-z^{-1})}{(1-\alpha z^{-1})}$$

This transfer function can be implemented with the following structure.

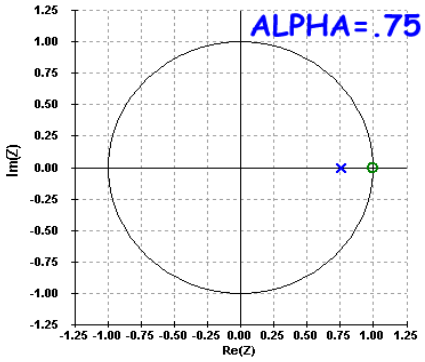


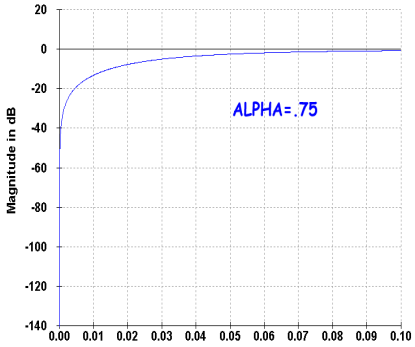

The code segment to implement is:

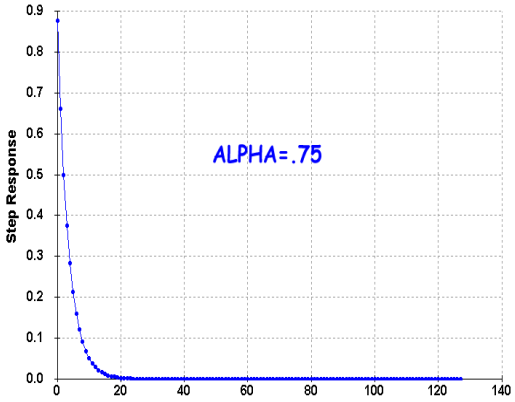
```
mag = sqrt(in.re*in.re + in.im*in.im);
w0 = mag + (m_w1 * ALPHA);
pOutData[i] = (w0 - m_w1);
m_w1 = w0;
```

The constant ALPHA can be adjusted to trade off step response(how long it takes to adjust out the DC) and the passband width. The closer ALPHA is to one, the longer it takes to settle but the low frequency response is better.

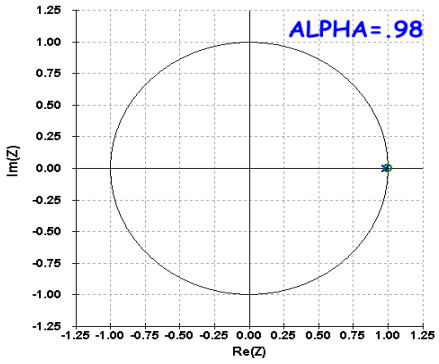
Here is the Pole-Zero plot, the Frequency Response, and Step Response using an ALPHA of .75.
Note the Step response takes <20 Samples to settle but the frequency response rolls off below .05*Sample Rate.

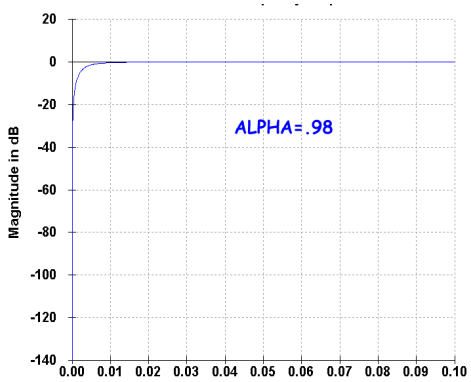


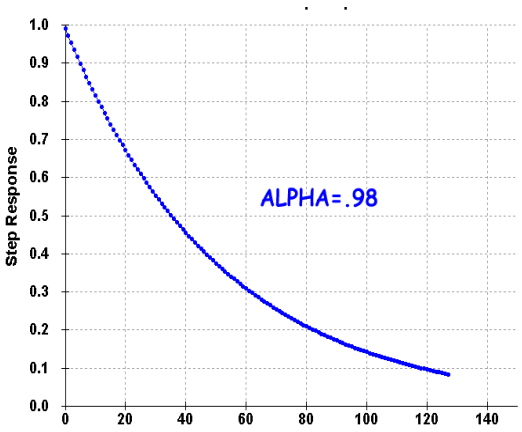



Here is the Pole-Zero plot, the Frequency Response, and Step Response using an ALPHA of .98.
Note the Step response takes >140 Samples to settle but the frequency response rolls off below .01*Sample Rate.

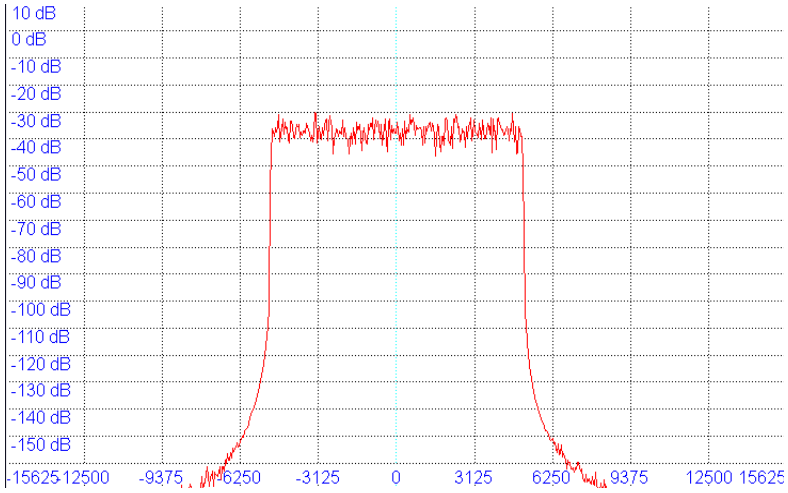


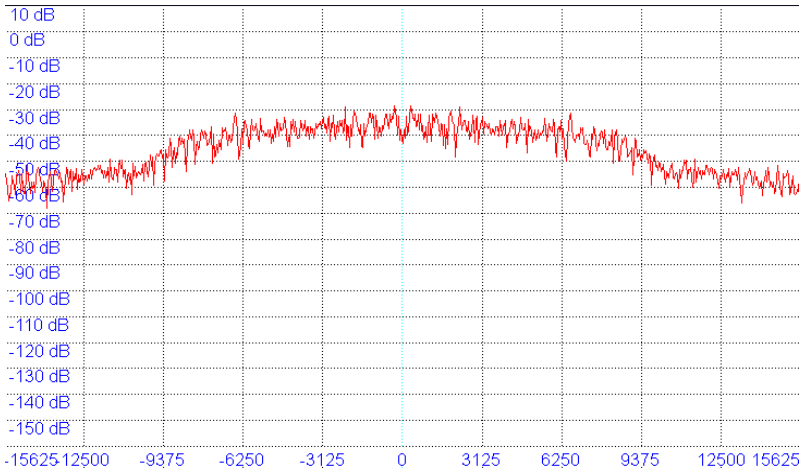



A secondary affect of the envelope detector is that as the signal drops into the noise, the output of the envelope detector creates frequencies outside of the input bandwidth.

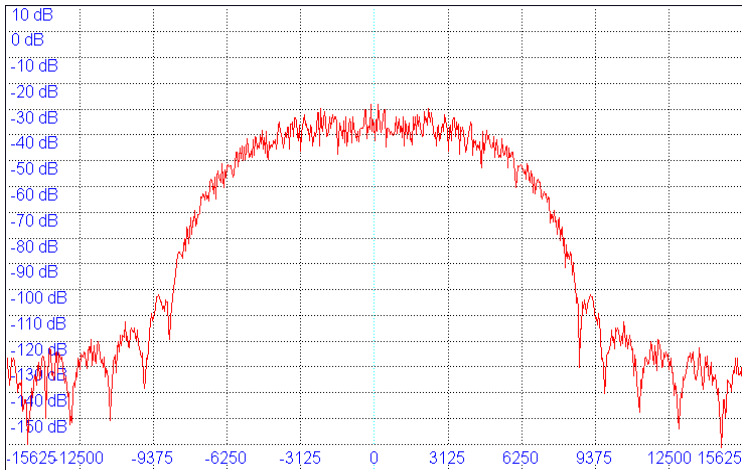
Here is the input to the AM detector with all noise showing the noise after filtering at +-5000Hz.




This is output of the AM envelope detector of the above input source. Note that noise is generated well outside the filter bandwidth.




By adding a low pass FIR filter after the envelope detector, the noise can be reduced outside the main filter bandwidth minimizing distortion on weak signals.



### 4.12.2 Implementation

The AM detector code segment is as follows.

```
for(int i=0; i<InLength; i++)
{
    //calculate instantaneous power magnitude of pInData which is I*I + Q*Q
    TYPECPX in = pInData[i];
    TYPEREAL mag = sqrt(in.re*in.re+in.im*in.im);
    //High pass filter(DC removal) with IIR filter
    // H(z) = (1 - z^-1)/(1 - ALPHA*z^-1)
    TYPEREAL z0 = mag + (m_z1 * DC_ALPHA);
    pOutData[i] = (z0 - m_z1);
    m_z1 = z0;
}
//post filter AM audio to limit high frequency noise
m_Fir.ProcessFilter(InLength, pOutData, pOutData);
```

The FIR low pass filter is a separate class that designs and implements a FIR filter and is described in a later section.

## *4.13. Synchronous AM Detector (CSamDemod Class)*

This class implements a synchronous AM detector. It takes the filtered I/Q data and outputs either stereo or mono demodulated output.

### 4.13.1 Design

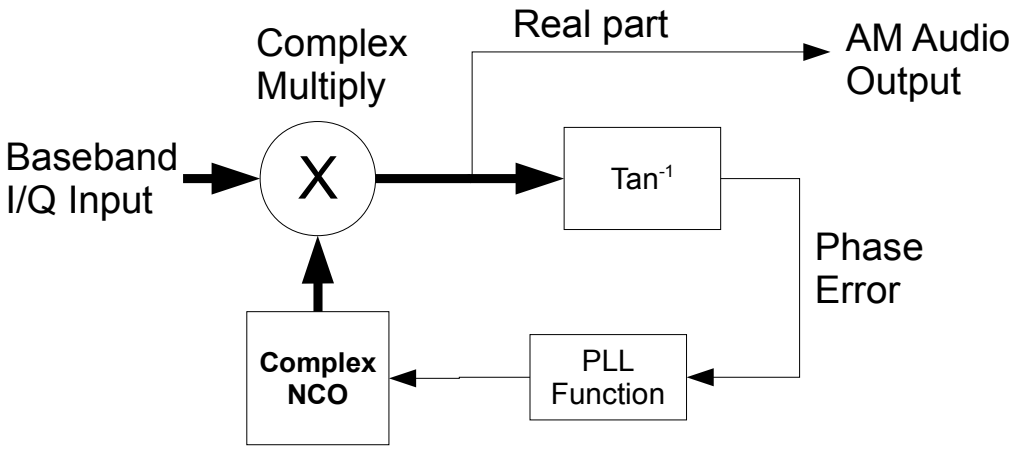
The following is the basic scheme noting the following as the equation of the AM signal.

$$e(t) = A_c[1 + mx(t)]\cos(\omega_c t + \theta_c)$$

If one shifts the carrier frequency to zero and the phase to zero or a constant, then the cos term is a constant and one is left with the desired signal plus a constant offset.

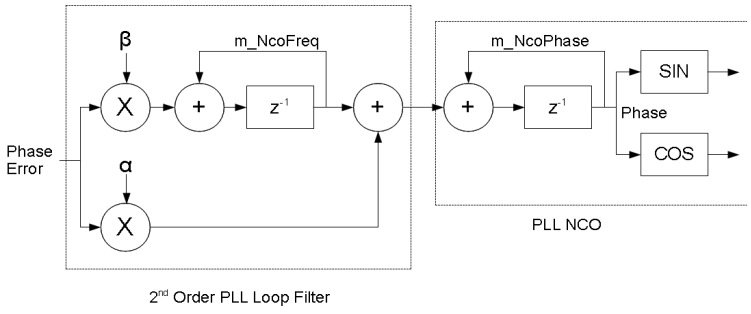
Complex multiply the input I/Q data by a complex NCO signal to shift the AM carrier to exactly zero frequency and a constant phase error. If this is done, then the real part of the output is just the AM demodulated audio plus the carrier amplitude which is just a DC offset. The advantage of this method is that it does not introduce the distortion that the envelope detector does in low signal to noise situations. Also since we are working with complex signals, one can independently select which sideband to use or have different filtering characteristics on each sideband to avoid interference.

The trick is how to generate the complex NCO mixer frequency to create the offset needed to meet this condition.




By taking the arctan of the resulting complex signal after mixing, a phase error signal can be generated that then can be manipulated to steer the NCO to the zero frequency error condition. From control theory, it requires a 2nd order PLL loop to lock with zero frequency error. A constant phase error will result.

Consider the following PLL function structure.



With a little work, the “open loop” transfer of the loop filter and NCO phase accumulator can be described in the z domain with a common (z-1) term as:

$$G(z)=\frac{\alpha(z-1)+\beta}{(z-1)^2}$$

The “closed loop” transfer function when the phase error is fed back and the loop is closed takes the form of:

$$H(z)=\frac{G(z)}{1+G(z)}$$

or

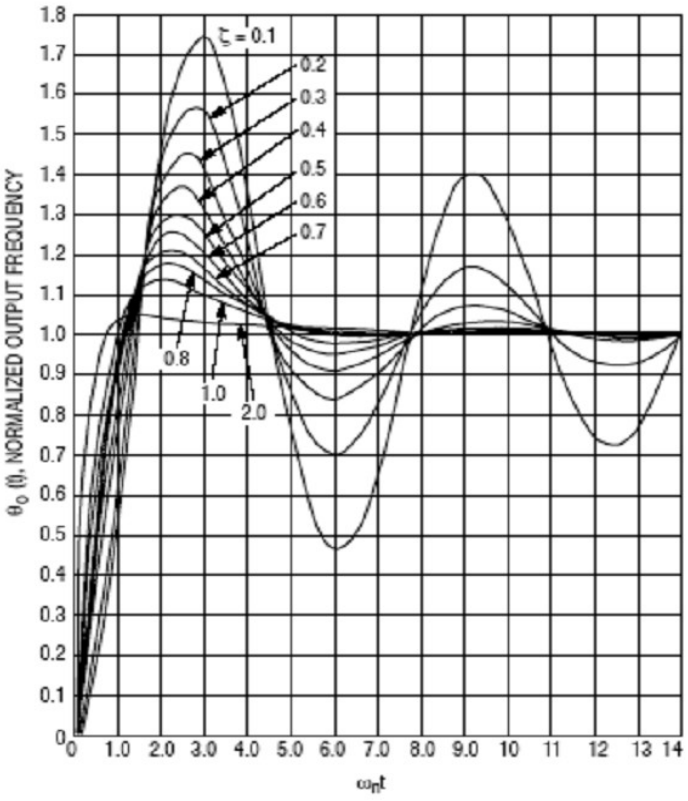
$$H(z)=\frac{\alpha(z-1)+\beta}{(z-1)^2+\alpha(z-1)+\beta}$$

Note that this form looks like the equivalent transfer function of a 2nd order analog PLL if you let s=(z-1).
With this in mind one can equate α and β in terms of natural frequency and damping factor.

Let: $\alpha=2\zeta\omega_n$ and $\beta=\omega_n^2$

$$H(z)=\frac{2\zeta\omega_n(z-1)+\omega_n^2}{(z-1)^2+2\zeta\omega_n(z-1)+\omega_n^2}$$

ζ sets the stability and transient response of the loop and $\omega_n$ sets the bandwidth of the loop.

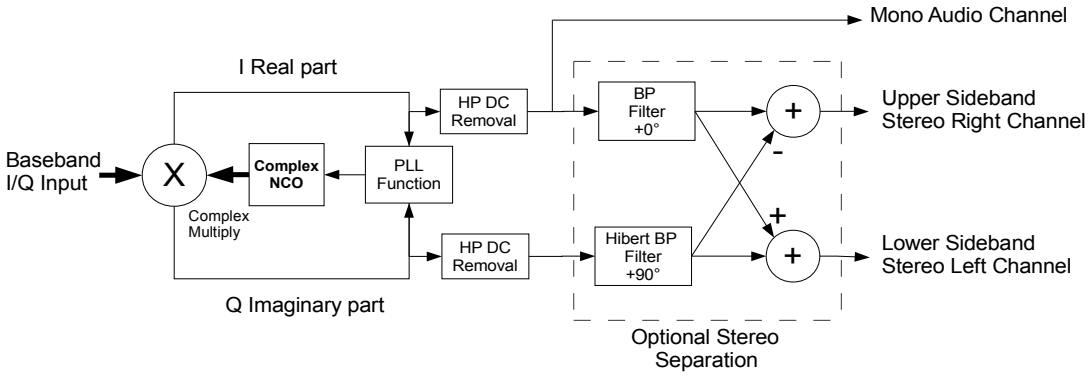


### 4.13.2 Implementation

The final configuration of the synchronous AM detector is shown below. The high pass DC removal filters are identical to the one described in detail for the envelope AM detector.

An optional stereo separation is implemented that separates the upper and lower sideband parts of the AM signal and sends to the stereo left and right channels. Normally the upper and lower sideband of an AM signal is the same but because of propagation distortion of the AM signal, the upper and lower sideband components can be different. By separating them into the left and right stereo channels, a pseudo stereo affect can be obtained giving some spatial enhancement of the signal.




In order to obtain both sideband signals from the I/Q baseband DSB AM signal, one must phase shift one of the I or Q channels by 90° degrees then adding or subtracting to obtain the desired sideband. This is the normal phasing method of SSB demodulation.

One way to create and I and Q data stream that has a 90 degree phase shift between them is to generate a low pass FIR filter then apply a transform on the filter coefficients to create a bandpass filter with 90 degree phase shift between them.

The FIR filter is designed and implemented in the CFir class and is described in detail in the CFir class section.

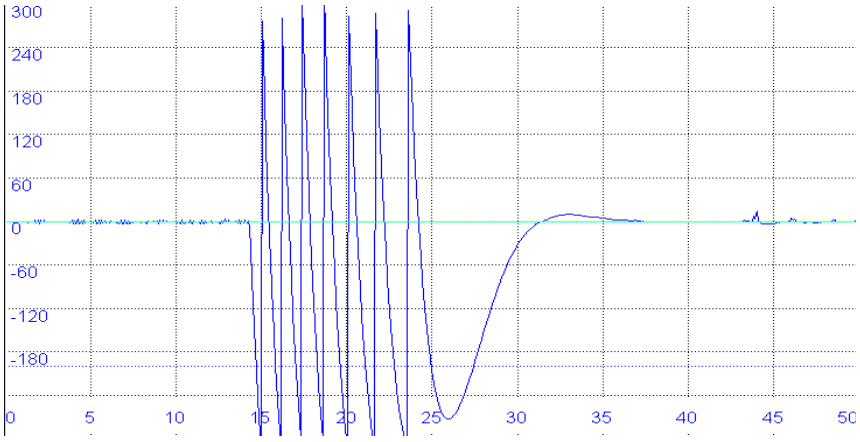
The stereo Synchronous AM detector is implemented with the following code segment.

```
for(int i=0; i<InLength; i++)
{
        TYPEREAL Sin = sin(m_NcoPhase);
        TYPEREAL Cos = cos(m_NcoPhase);
        //complex multiply input sample by NCO's  sin and cos
        tmp.re = Cos * pInData[i].re - Sin * pInData[i].im;
        tmp.im = Cos * pInData[i].im + Sin * pInData[i].re;
        //find current sample phase error after being shifted by NCO frequency
        TYPEREAL phzerror = -atan2(tmp.im, tmp.re);
        m_NcoFreq += (m_PllBeta * phzerror);          //  radians per sampletime
        //clamp NCO frequency so doesn't drift out of lock range
        if(m_NcoFreq > m_NcoHLimit)
                m_NcoFreq = m_NcoHLimit;
        else if(m_NcoFreq < m_NcoLLimit)
                m_NcoFreq = m_NcoLLimit;
        //update NCO phase with new value
        m_NcoPhase += (m_NcoFreq + m_PllAlpha * phzerror);
        //High pass filter(DC removal) with IIR filter
        // H(z) = (1 - z^-1)/(1 - ALPHA*z^-1)
        TYPEREAL z0 = tmp.re + (m_z1 * DC_ALPHA);
        TYPEREAL y0 = tmp.im + (m_y1 * DC_ALPHA);
```

```
            pOutData[i].re = (z0 - m_z1);
            pOutData[i].im = (y0 - m_y1);
            m_y1 = y0;
            m_z1 = z0;
}
    m_NcoPhase = fmod(m_NcoPhase, K_2PI);   //keep radian counter bounded
    //process I/Q with bandpass filter with 90deg phase shift between the I and Q filters
    m_Fir.ProcessFilter(InLength, pOutData, pOutData);
for(int i=0; i<InLength; i++)
{
    tmp = pOutData[i];
    pOutData[i].im = tmp.re - tmp.im;       //send upper sideband to (right)channel
    pOutData[i].re = tmp.re + tmp.im;       //send lower sideband to (left)channel
}
```

### 4.13.3 Design Verification

The step response of the PLL can be verified by quickly changing the demod frequency by 1KHz up and down and plotting the PLL error signal versus time. The PLL parameters for AM were set to 100Hz bandwidth, and damping factor ζ= .707.



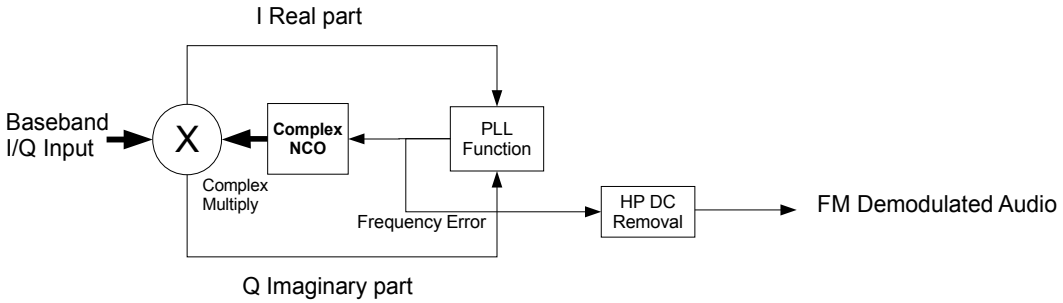

The seven or so cycles of error is while the frequency is changing and the phase error is greater than +-π. The final lock waveform then resembles the normal 2nd order damping shape with a slight overshoot. The ringing is close to the period of the 100Hz bandwidth.

## *4.14. FM Demodulation (CFmDemod Class)*

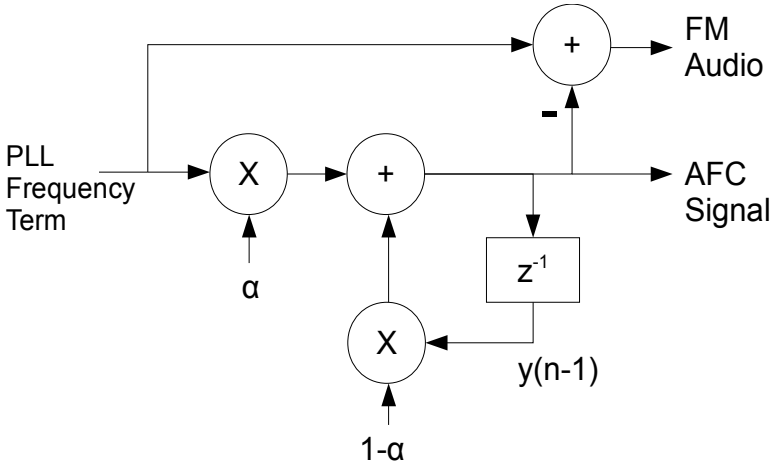
### 4.14.1 Design

FM demodulation can be achieved in various ways such as implementing the analog equivalent of a discriminator detector. CuteSDR uses a PLL identical in structure to the Synchronous AM detector described earlier. The difference is where the demodulated audio is extracted. In the SAM version the real part of the I/Q data stream was the desired output. For FM, one takes the frequency term from the PLL feedback path. As the PLL tracks the FM signal, it generates a signal that is the instantaneous frequency of the incoming I/Q baseband signal.

The basic scheme is shown below.




One must change the PLL parameters from the SAM version since it must now track an FM signal. The PLL bandwidth must be high enough to pass the demodulated signal bandwidth. If the FM signal is slightly off frequency, the frequency error will contain a DC term which can be used for an AFC function but must be removed from the audio. If one filters the frequency error with an exponential averager, the AFC term can be extracted and if this same value is subtracted from the frequency error, the DC component is removed.




Exponential Averaging Filter

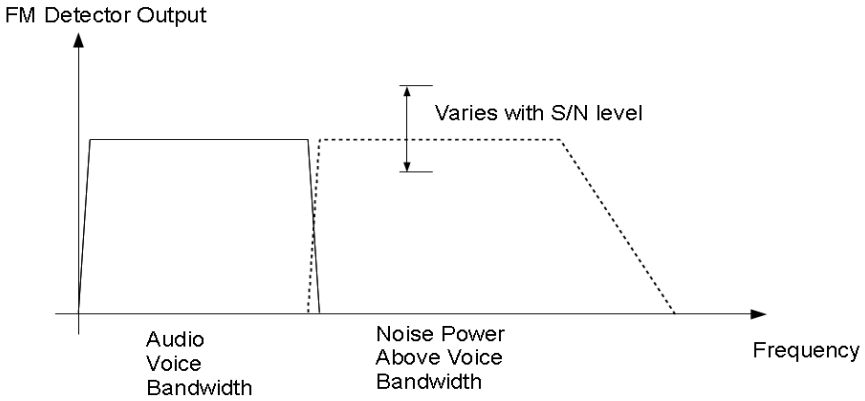
### 4.14.2 FM Demodulator Implementation

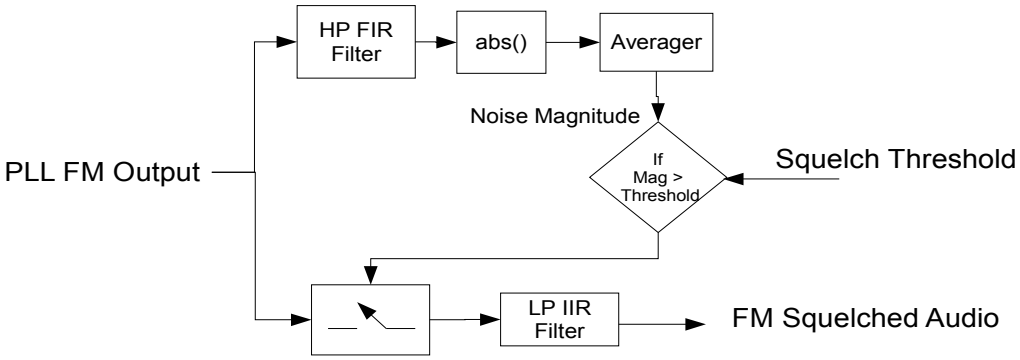
Below is a segment of the FM demod PLL code:

```
for(int i=0; i<InLength; i++)
{
    TYPEREAL Sin = sin(m_NcoPhase);
    TYPEREAL Cos = cos(m_NcoPhase);
    //complex multiply input sample by NCO's  sin and cos
    tmp.re = Cos * pInData[i].re - Sin * pInData[i].im;
    tmp.im = Cos * pInData[i].im + Sin * pInData[i].re;
    //find current sample phase after being shifted by NCO frequency
    TYPEREAL phzerror = -atan2(tmp.im, tmp.re);
    //create new NCO frequency term
    m_NcoFreq += (m_PllBeta * phzerror);           //  radians per sampletime
    //clamp NCO frequency so doesn't get out of lock range
    if(m_NcoFreq > m_NcoHLimit)
            m_NcoFreq = m_NcoHLimit;
    else if(m_NcoFreq < m_NcoLLimit)
            m_NcoFreq = m_NcoLLimit;
    //update NCO phase with new value
    m_NcoPhase += (m_NcoFreq + m_PllAlpha * phzerror);
    //LP filter the NCO frequency term to get DC offset value
    m_FreqErrorDC = (1.0-m_DcAlpha)*m_FreqErrorDC + m_DcAlpha*m_NcoFreq;
    //subtract out DC term to get FM audio
    pOutData[i] = (m_NcoFreq-m_FreqErrorDC)*m_OutGain;
}
```

### 4.14.3 FM Noise Squelch Design

Another useful function for an FM demodulator is a squelch feature. One could use the signal power level or use a characteristic of FM detectors where the noise level decreases with the strength of the input signal. CuteSDR implements a noise squelch with a high pass filter that measures noise power higher than the FM audio bandwidth of a voice signal and mutes the audio if the noise magnitude is above a user selected threshold.



### 4.14.4 FM Noise Squelch Implementation

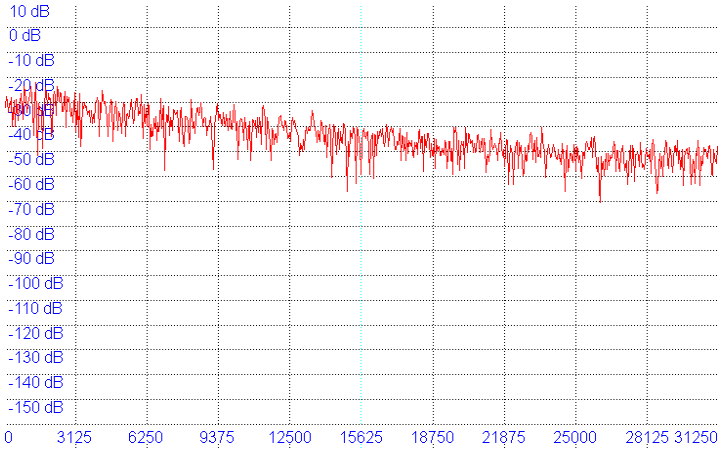
The high pass filter is implemented with a FIR filter with a cutoff frequency equal to the input filter bandwidth. A simple low pass IIR filter is used to roll off the audio from about 3KHz to reduce high frequency audio noise.

Below is a code segment from the noise squelch function:

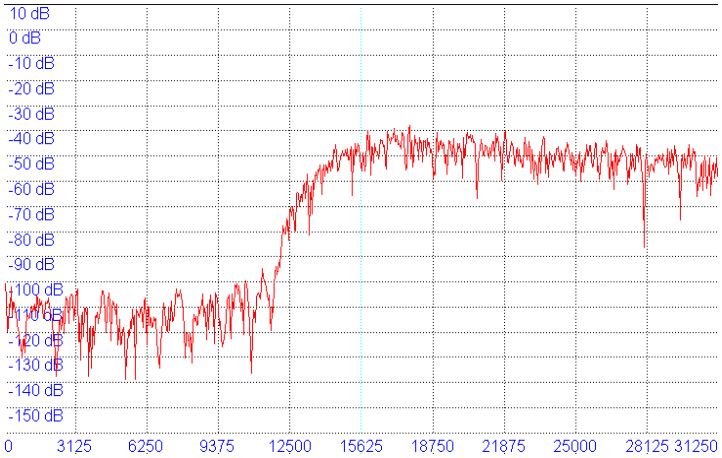
```
TYPEREAL sqbuf[MAX_SQBUF_SIZE];
//high pass filter to get the high frequency noise above the voice
m_HpFir.ProcessFilter(InLength, pOutData, sqbuf);
for(int i=0; i<InLength; i++)
{
	TYPEREAL mag = fabs( sqbuf[i] );	//get magnitude of High pass filtered data
	// exponential filter squelch magnitude
	m_SquelchAve = (1.0-m_SquelchAlpha)*m_SquelchAve + m_SquelchAlpha*mag;
}
//perform squelch compare to threshold using some Hysteresis
if(0==m_SquelchThreshold)
{	//force squelch if zero(strong signal threshold)
	m_SquelchState = true;
}
else if(m_SquelchState) //if in squelched state
{
	if(m_SquelchAve < (m_SquelchThreshold-SQUELCH_HYSTERESIS))
		m_SquelchState = false;
}
else
{
	if(m_SquelchAve >= (m_SquelchThreshold+SQUELCH_HYSTERESIS))
		m_SquelchState = true;
}
if(m_SquelchState)
{	//zero output if squelched
	for(int i=0; i<InLength; i++)
		pOutData[i] = 0.0;
}
else
{	//low pass filter audio if squelch is open
	m_Lplir.ProcessFilter(InLength, pOutData, pOutData);
}
```

### 4.14.5 FM Demodulator/Squelch Testing

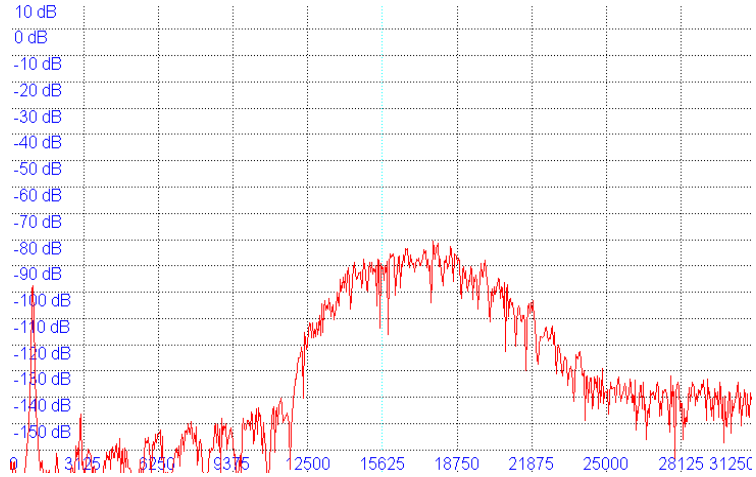
This is the output of the FM PLL detector with a +-15KHz input filter and just noise. Note the almost flat spectrum with lots of energy outside the 3KHz audio band. This is what is used to obtain a squelch signal.




This is the same signal but after the Squelch high pass filter.

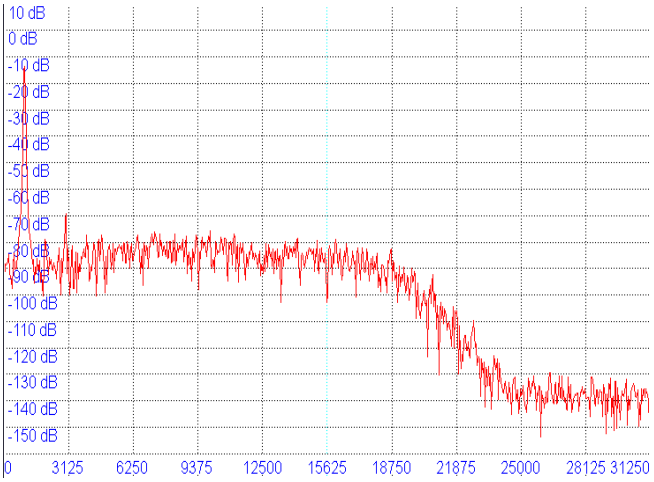



This is the output of the high pass filter but with a strong 1KHz FM signal. Note the large decrease in power due to the presence of an FM signal. The 1KHz audio signal is present but attenuated by the HP filter.

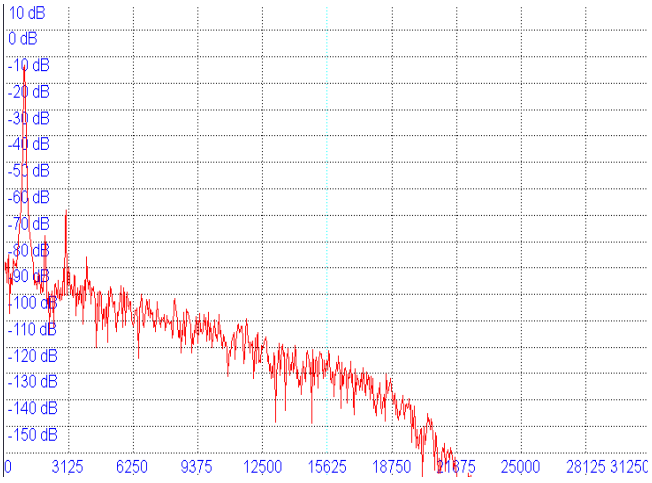


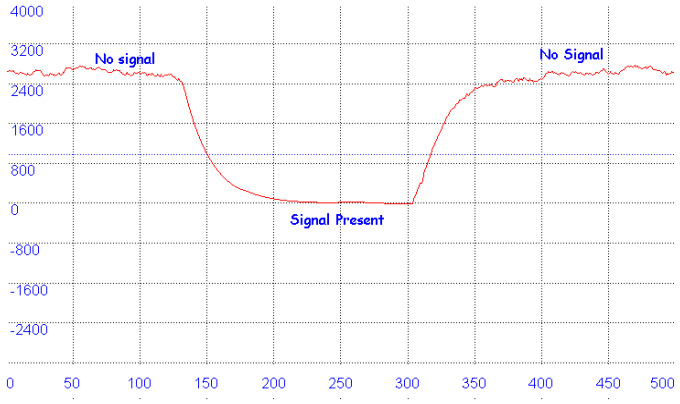
This is the same 1KHz FM signal but from the raw PLL detector. While clean, there is still energy above the 3KHz audio bandwidth. The third harmonic of the 1KHz signal can be seen and is probably from the HP8657A signal generator that was providing the signal.




The final audio output after the low pass IIR filter shows the roll off above 3KHz.



Squelch operation is shown here with an FM signal transitioning from noise to signal to noise again.



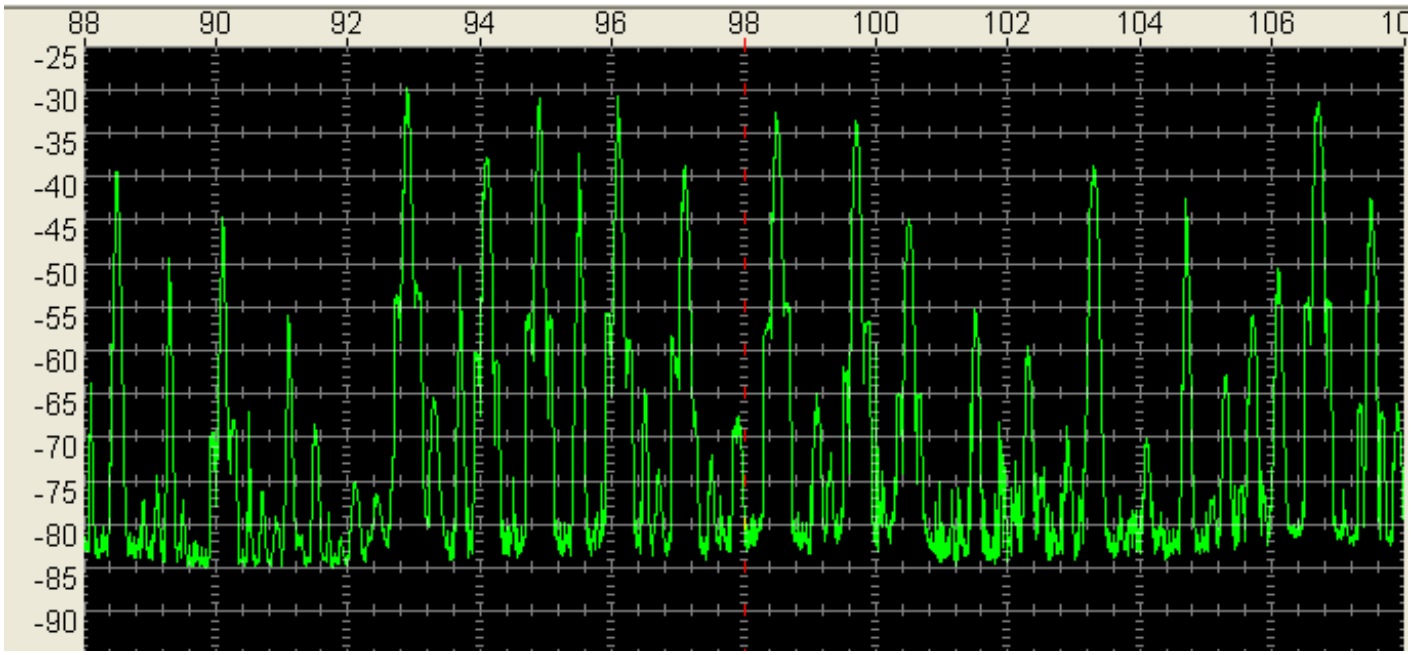
## *4.15. WFM Demodulation (CWFmDemod Class)*

### 4.15.1 Broadcast FM Signals

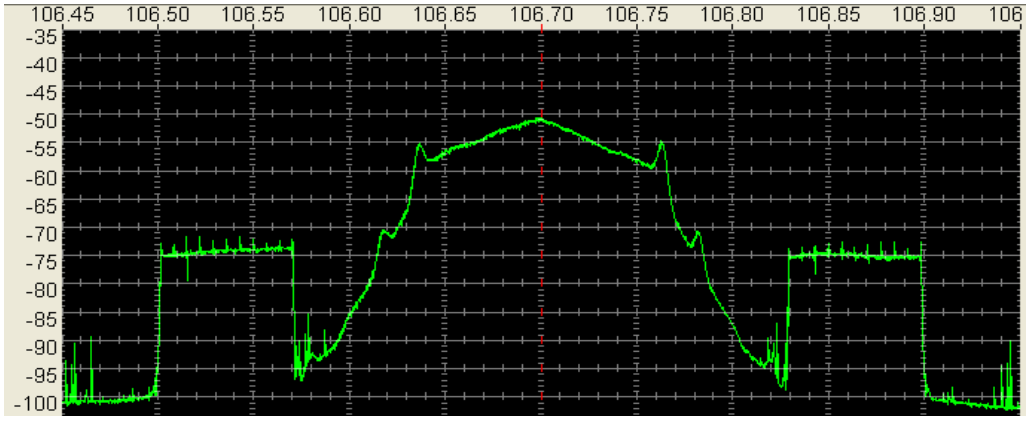
The first task in designing a wide band FM demodulator for the broadcast band is to understand how the FM signal is formed and decide what performance is desired and what ancillary signals are needed.

The FM signal itself is just FM modulation and behaves similarly to narrow band FM except it is much wider. In the US, the RF band covers 87.8 MHz to 108.0 Hz with a frequency deviation of around +-75KHz in a channel 200KHz wide. US hybrid stations are twice as wide using the extra bandwidth to include a proprietary digital modulation outside the 200KHz analog FM signal.

The following shows the 88 to 108MHz FM band spectrum of a typical metro area.

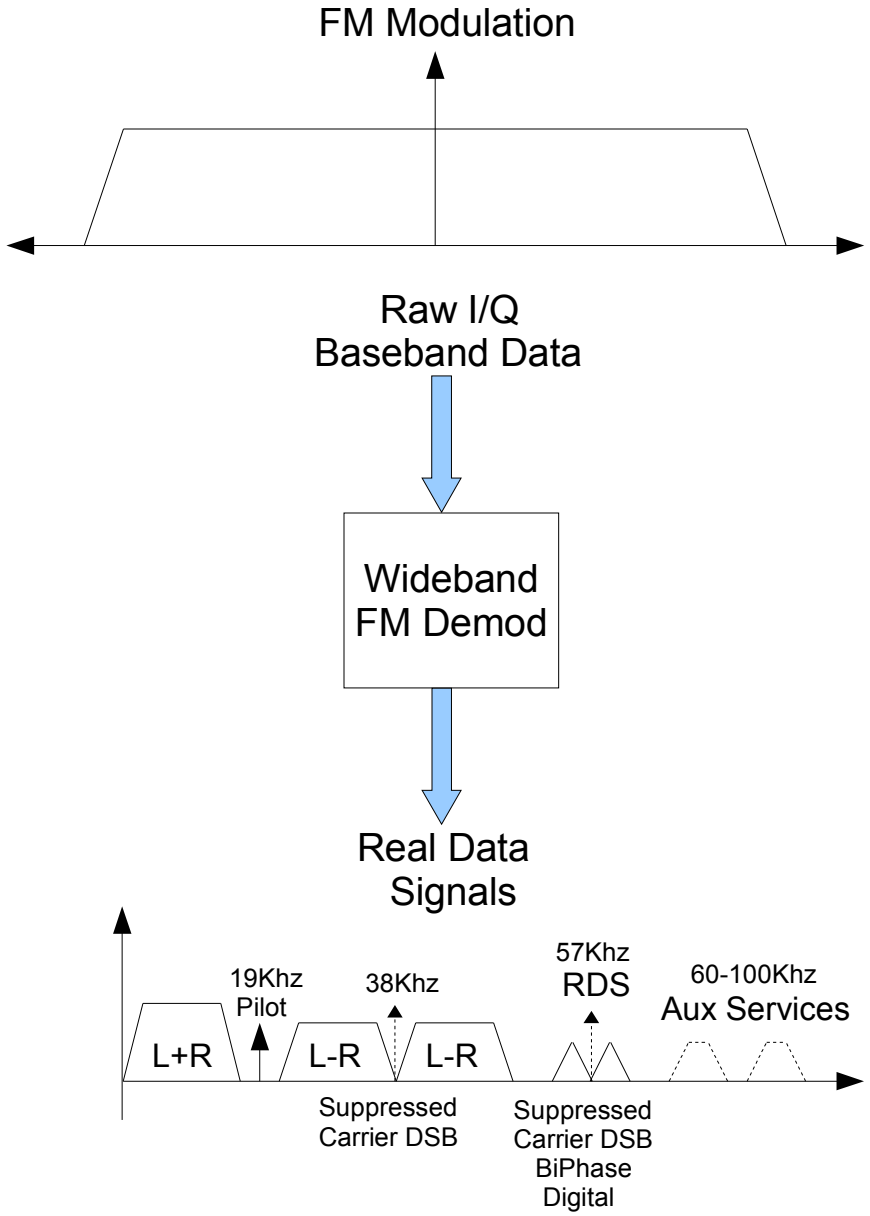



This is a close up of a typical hybrid FM station with the distinguishable square digital side lobes.



One way of looking at FM broadcast signals is that there is another spectrum of assorted signals contained in the demodulated primary FM signal. One can then treat the demodulated FM signal as a completely new signal containing various analog and digital information that can be demodulated separately.



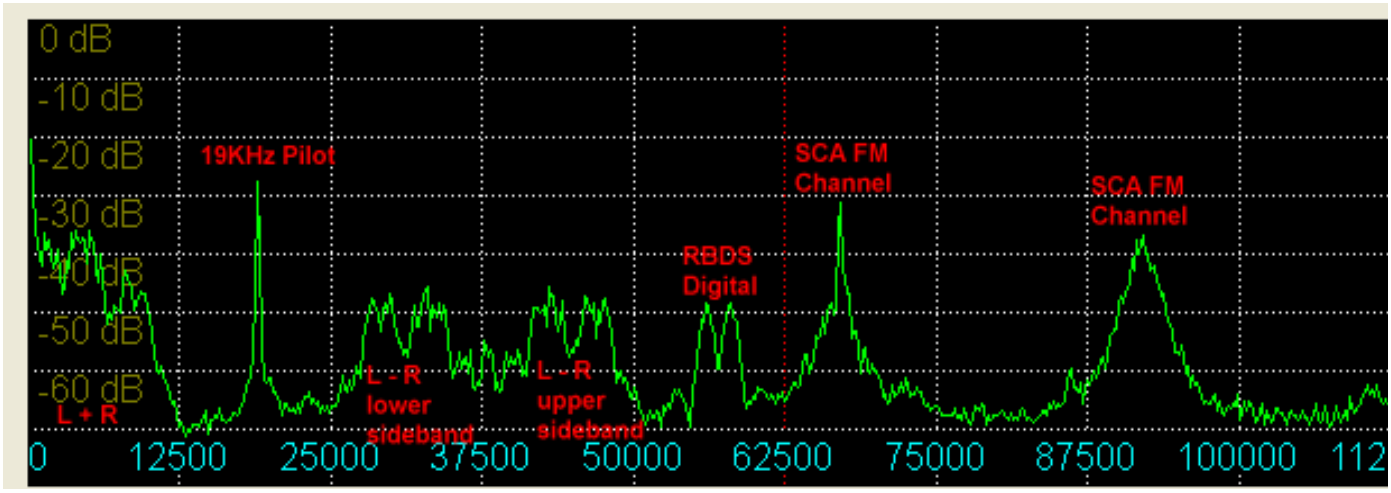

The monaural FM audio signal comes out directly from the demodulated FM from 0 to 15KHz. When the station is in stereo mode, the left plus the right channel is the same as the monaural signal from 0 to 15KHz. A separate audio signal consisting of the left minus the right channel is shifted by 38KHz as a suppressed carrier double sideband signal occupying from around 23 to 53KHz of the demodulated spectrum. A pilot tone at 19KHz is then added which can be locked to and used to recover the DSB Left minus Right audio at 38KHz.

An optional RDS digital signal can be found at 57KHz and is another suppressed carrier double sideband modulated signal with the modulation source being a bi-phase digital signal containing various items depending on the station.

Other optional signals can be found above 60KHz. Some are narrow band FM analog signals with alternate language talk radio or other voice uses. Other proprietary digital services can also be found in this region.

Below is a typical spectrum of an FM demodulated station containing all the above signals.

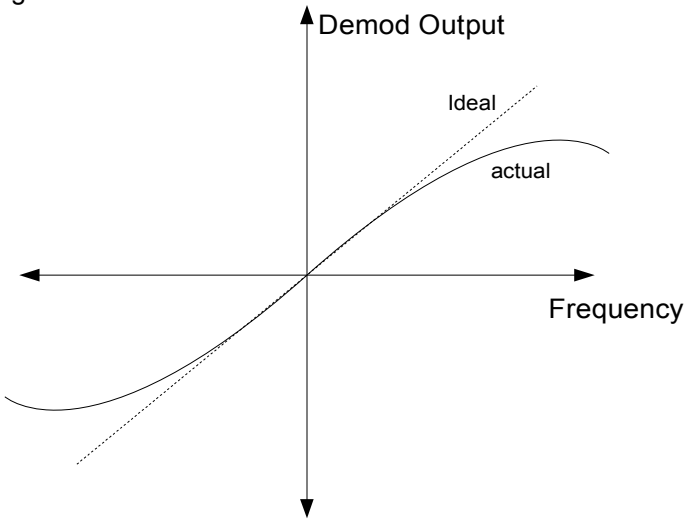



This CWFmDemod class was designed to extract the stereo audio as well as the RDS data blocks if present. The SCA channels were ignored due to their limited usefulness or proprietary encoding.

### 4.15.2 Demodulation Process

Prior to this demodulator stage, the I/Q data stream is decimated into a range from about 250KHz to 500KHz from whatever the radio input rate is. The first stage performs sideband FM demod on the incoming I/Q data. Several methods for FM demod were considered. Since this stage runs at a fairly high sample rate compared to all the other demodulator stages, an efficient algorithm is needed.

Since the primary use of broadcast FM is to transmit high fidelity audio, the main goal for the demodulator is low distortion. This means the transfer function of the demodulator should be as close to an ideal straight line as possible. Any variation from the ideal straight line leads to distortion.




FM demodulation can be done by finding the phase changes of the I/Q signal then taking the derivative of the phase changes to get frequency. A PLL can also be used just as was done for the narrow band FM demodulator.

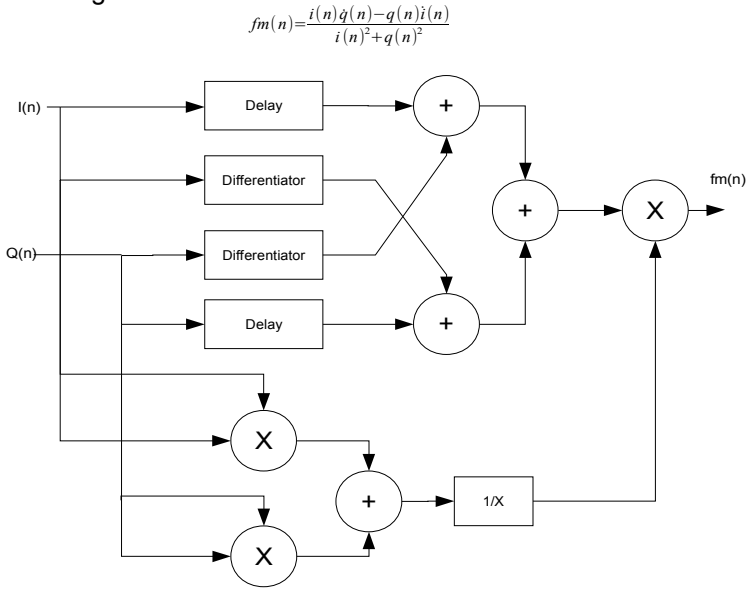
Various ways can be used to implement the FM detector and we will look at three methods and analyze them for suitability.

### 4.15.2.a Dual Differentiator Method

One method is to use the following scheme that takes advantage of a trig identity that calculates the derivative of the arctan() function directly from the I/Q data stream without using any trig functions.

$$fm(n)=\frac{i(n)\dot{q}(n)-q(n)\dot{i}(n)}{i(n)^2+q(n)^2}$$ Where $\dot{i}(n)$ and $\dot{q}(n)$ are the derivatives of I and Q.

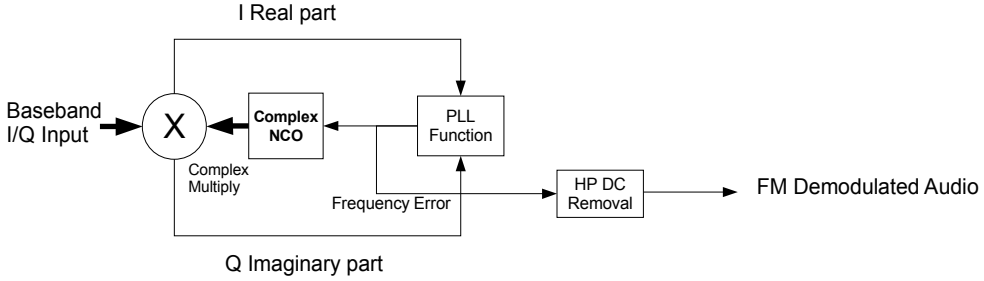
In block form the algorithm is as follows:




The critical element in this scheme is the Differentiator block. In a sampled data system the implementations of the differentiator are non-exact and as a result have non-linear regions.

This detector works fairly well for communications grade FM but since broadcast FM is meant to be high quality this scheme suffers from high distortion due to the non linearity of the differentiator blocks. Even with elaborate differentiator filters, about 40dB harmonic distortion levels were the best that could be obtained so this method was scrubbed.

### 4.15.2.b PLL Method

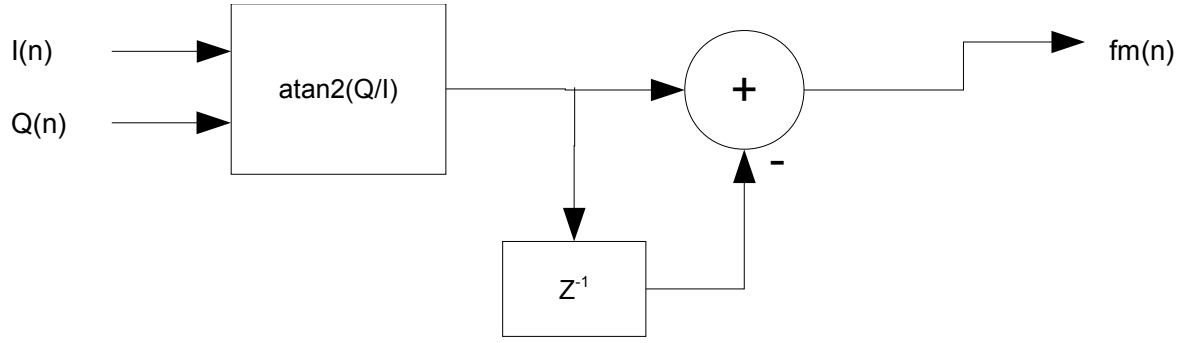



A PLL such as that used in the narrow band FM detector was tried and while it does work, it required much higher oversampling rates to remain linear and recover the stereo components. They also use a lot of CPU cycles due to needing a sin/cos NCO and arctan phase detector.

4.15.2.c ARCTAN Method

The straight forward approach and the one that was used in CuteSDR is to take the arctan of each sample and subtract from the previous samples arctan to get the rate of phase change or frequency.




The problem with this method is dealing with the boundaries of the arctan function and figuring out the actual angle difference between successive samples.

One thing to do is use the atan2(y,x) math function instead of the atan(y/x) function. This helps because the boundaries are taken care of by the atan2() function.

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \pi + \arctan\left(\frac{y}{x}\right) & y \geq 0, x < 0 \\ -\pi + \arctan\left(\frac{y}{x}\right) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

The differentiator following the arctan function can also create problems again due to having to deal with what quadrant the points are in and if they cross the boundaries of +/-π or a point is on the origin creating a divide by zero situation. The differential simply takes the phase angle difference between the current and previous sample to obtain frequency.

Let the 2 points be ($I_n$ , $Q_n$ ) and ($I_{n-1}$ , $Q_{n-1}$ ).

In polar notation the current sample and the complex conjugate of the previous sample are:

$M_n e^{j\theta_n} \quad M_{n-1} e^{-j\theta_{n-1}}$

If we multiply these together we get the following:

$M_n M_{n-1} e^{j(\theta_n - \theta_{n-1})}$

Note that this new vector has an angle that is the difference between the current and previous sample. Taking the arctan of this new vector will give the angle of the difference which is exactly what we want without having to do a subtraction after the arctan function which is messy.

The down side is a complex multiply of the current and the conjugate of the previous sample must be done.

The result of the complex multiply in rectangular coordinates is the following:

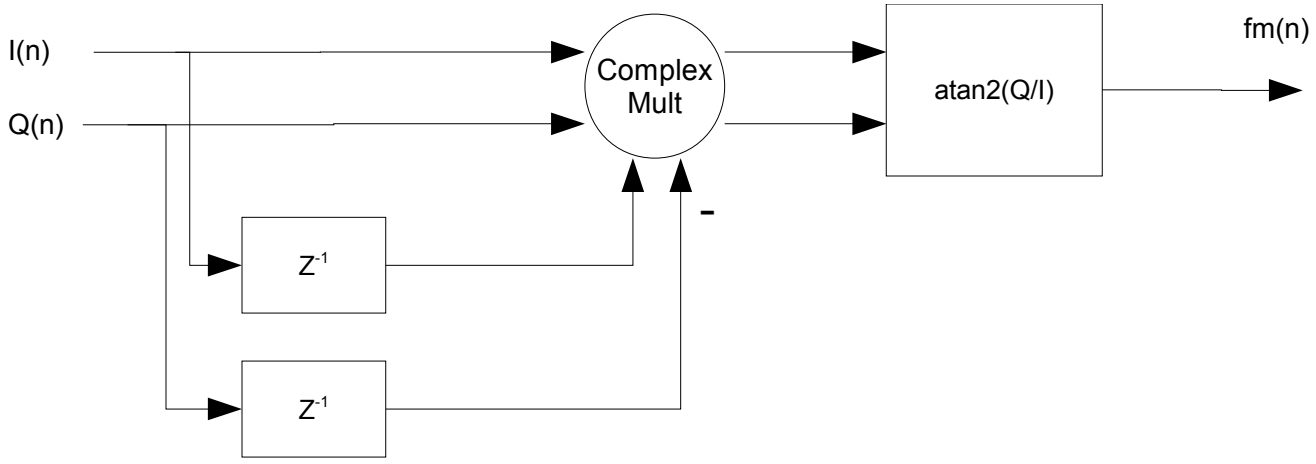
$$(I_n + jQ_n)(I_{n-1} - jQ_{n-1}) = (I_n i_{n-1} + Q_n Q_{n-1}) + j(I_{n-1} Q_n - I_n Q_{n-1})$$

Putting this new vector into the atan2() function gives the instantaneous frequency which is what we are after.

$$FM(n) = atan2\left(\frac{I_{n-1}Q_n - I_nQ_{n-1}}{I_n i_{n-1} + Q_n Q_{n-1}}\right)$$

In block diagram form the algorithm looks like this:




The source code snippet to implement this is:

```
m_D0 = pInData[i];
m_RawFm[i] = FMDEMOD_GAIN * atan2(   (m_D1.re*m_D0.im - m_D0.re*m_D1.im),
                                     (m_D1.re*m_D0.re + m_D1.im*m_D0.im) );
```

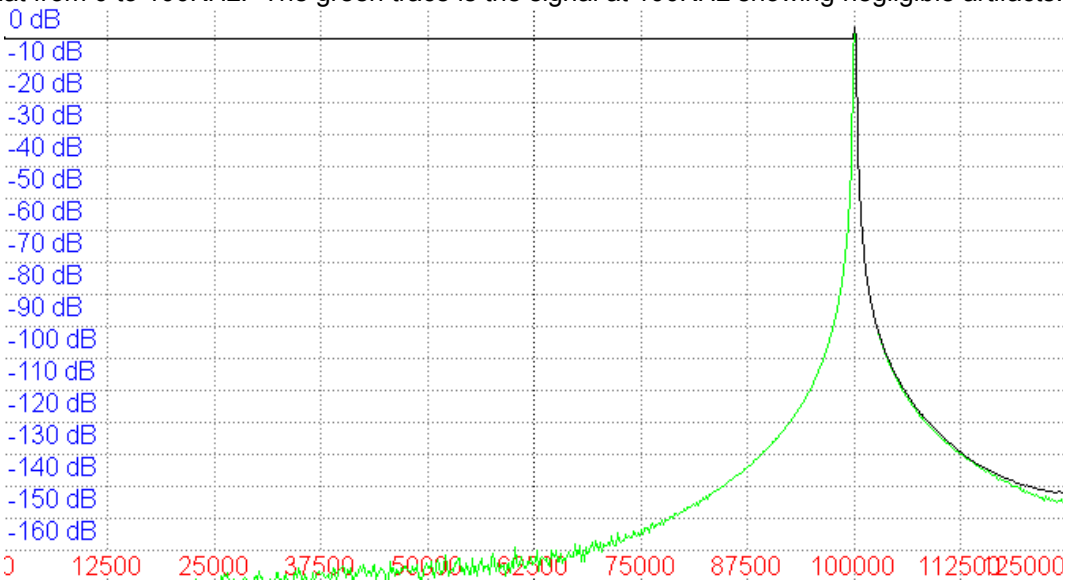
  

```
m_D1 = m_D0;
```

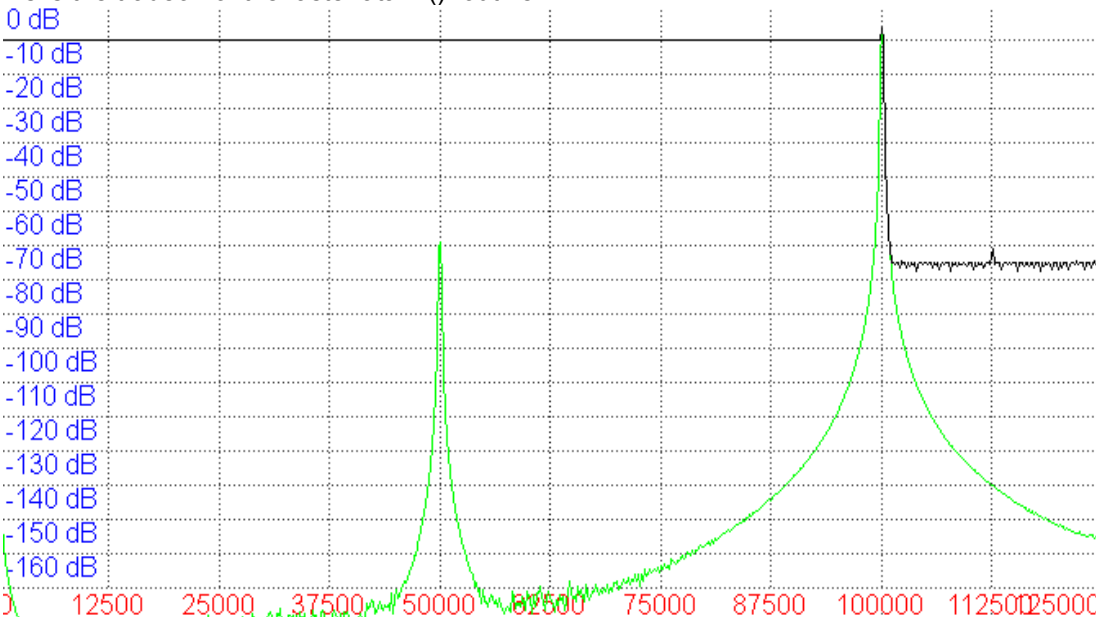
The atan2 function is a little expensive in terms of CPU cycles but with modern PC's it is acceptable. A lower resolution but faster algorithm was implemented in CuteSDR. The tradeoff is more distortion on the order of 60dB which is probably acceptable since there are worse sources of distortion such as from filtering and multipath.

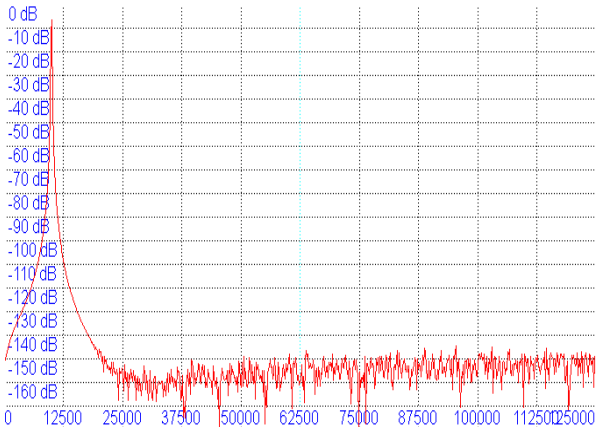
To test the algorithm, a sweep generator was created to modulate a complex FM generator. Using double precision math for both the generator and this demodulator without any filtering in between provided a nearly perfect response with harmonic distortion essentially non-existent. The following plot shows a sweep from 0 to 100KHz of a pure sine wave modulated FM signal with 75KHz deviation and a sample rate of 250KHz. The black line is the peak response as it sweeps and is flat from 0 to 100KHz. The green trace is the signal at 100KHz showing negligible artifacts.

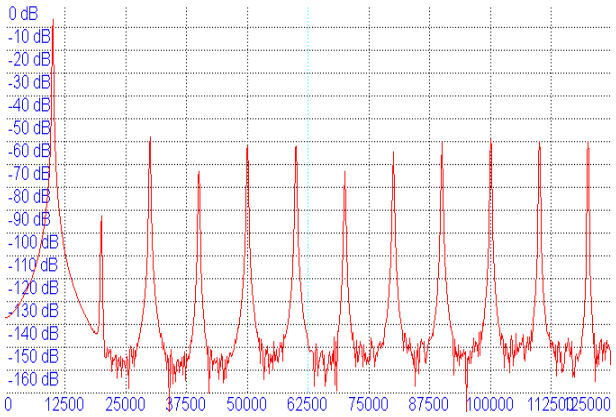



The same test was run using the faster but less accurate atan2() function. Note the spurious product 60dB below the main signal. This is the tradeoff of the faster atan2() routine.



Another point to make is that the FM signal is not contained in a finite bandwidth such as the case with AM or sideband. A pure sine wave modulated FM signal has many frequency components extending to infinity although they do drop in amplitude. As a result, if one filters these higher frequency components, the recovered FM signal will not be complete and thus be distorted. Below is a sweep from 0 to 100KHz as before but with and without a 220kHz IIR filter ahead of the demodulator stage. A tradeoff must be made between distortion and adjacent channel rejection and/or alias products.

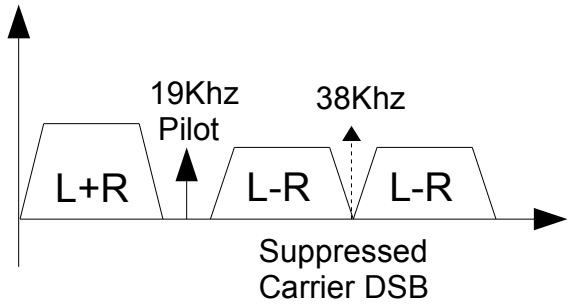


### 4.15.3 Stereo Decoding

In order to understand how to recreate the stereo audio from the demodulated FM signal, it is good to see how the signal is generated.

The basic equation for generating the stereo signal modulation is shown below.




$$m(t)=[\ 0.9[\frac{L+R}{2}+\frac{L-R}{2}\ \sin(4\pi f_p t)]\ +\ 0.1\sin(2\pi f_p)]$$

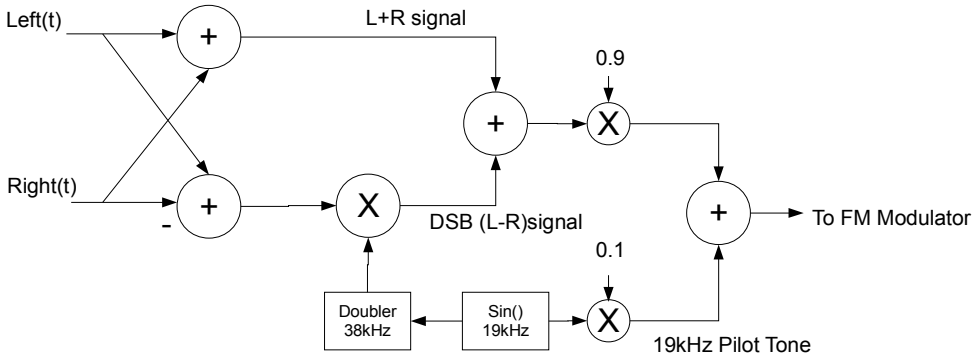
Where $f_p$ == pilot frequency 19kHz
L == Left audio signal
R == Right audio signal

The rightmost part of the equation generates a 19KHz pilot tone with a 10% amplitude.

The left part of the equation is the R+L signal and then the L-R signal multiplied by a 38Khz carrier(2 times the pilot tone). The amplitude of this is part is 90% of the total modulation.

By multiplying the L-R term by the sin of 38kHz, a double sideband suppressed carrier is generated around 38kHz.

In block diagram form it looks like this:



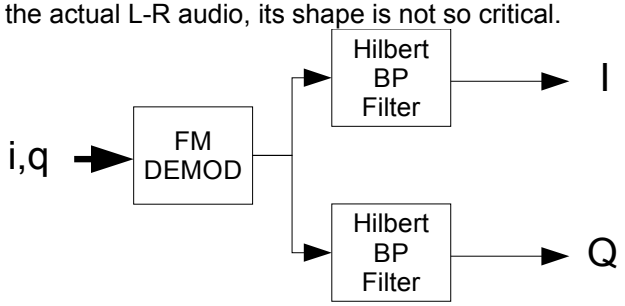
The C code snippet to generate a sample is here:

```
m_PilotAcc += m_PilotInc;       //create pilot freq
mod = ( m_Left + m_Right)/2.0;   //mono component
mod += (  (m_Left - m_Right)/2.0) * sin(m_PilotAcc*2.0 ); //DSB left-right component
mod *= 0.9;
mod += ( 0.1*sin(m_PilotAcc) );  //add pilot tone
```

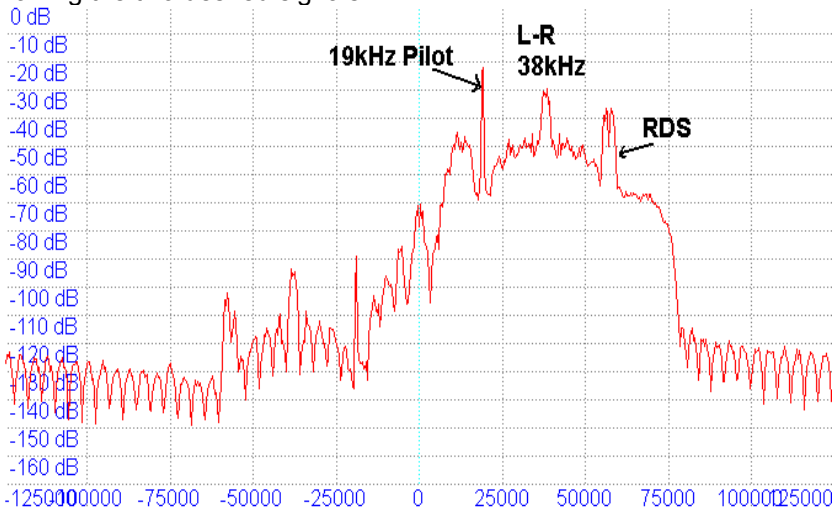
To demodulate the stereo signal one must reverse this process. The L+R signal is just the direct FM demodulated signal from 0 to 15kHz. In order to retrieve the L-R part, one must regenerate the 38kHz sine wave from the pilot tone at 19kHz.

Keeping in mind that other signals such as RDS and SCA signals may be desired to be demodulated, it was decided to create a complex base band signal from the real signal that comes from the FM demodulator. This would allow easier re-use of complex PLL blocks, filters, frequency shifters etc. If only the stereo signal was desired, it would probably be easier to remain in the real domain and use other techniques for generating the 38kHz signal.

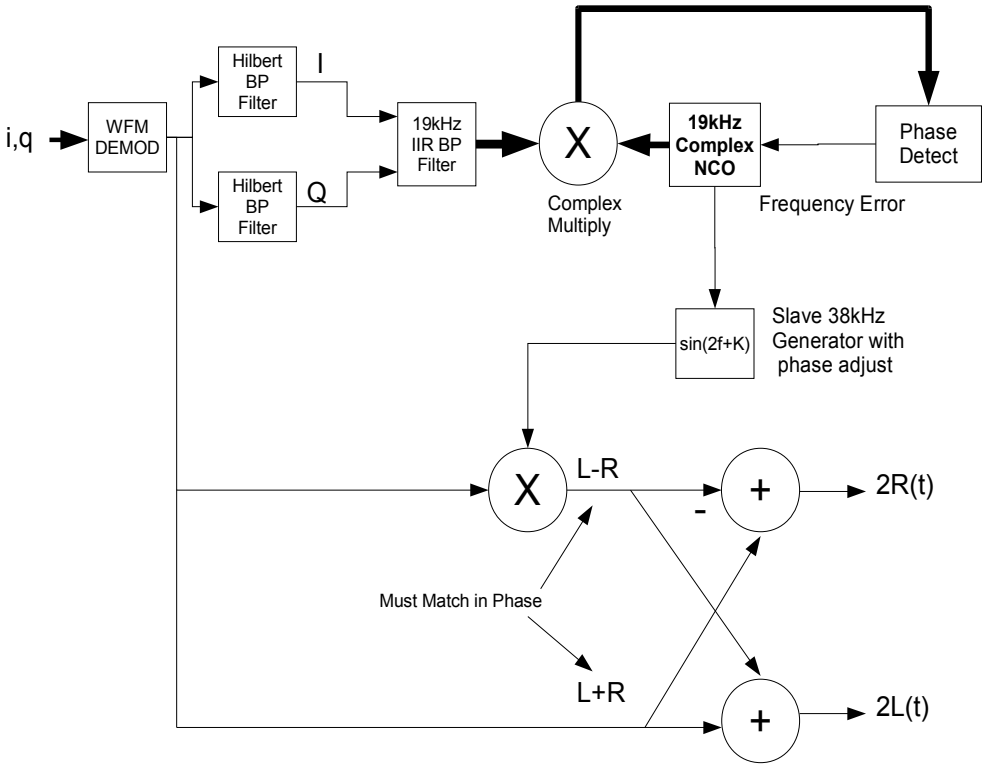
The Hilbert filter pair method was used to create a Hilbert bandpass filter covering from about 10kHz to 75kHz or so. (See section on “Hilbert Filter Pair Generation” for details). Since this complex signal will be used to just generate the 38kHz carrier and not be used for the actual L-R audio, its shape is not so critical.




Complex spectrum showing the two desired signals.



The following block diagram shows the stereo recovery scheme used. The complex signal is first bandpass filtered using an IIR filter then to the PLL to lock to the 19Khz pilot tone. Since the 19KHz pilot is very narrow and at a known frequency, a high Q filter can be used(Q=200). This narrow filter plus a narrow bandwidth PLL will easily lock to the stereo pilot. A lock detect output from the PLL is used to revert to monaural mode if there is no pilot to lock to.




The code segment that implements this is:

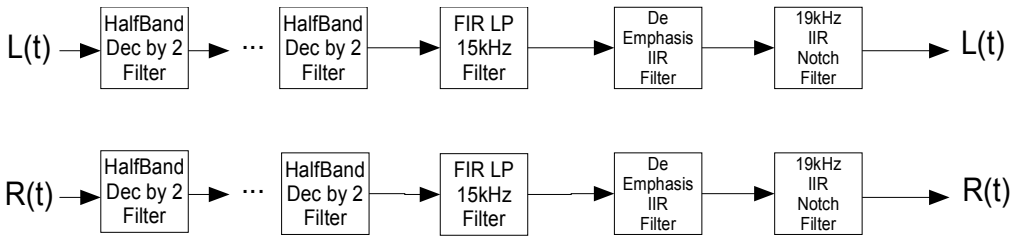
```
//create complex data from demodulator real data
m_HilbertFilter.ProcessFilter(InLength, m_RawFm, m_CpxRawFm);
m_PilotBPFilter.ProcessFilter(InLength, m_CpxRawFm, pInData);    //use input buffer for complex output storage
if(ProcessPilotPll(InLength, pInData) )
{      //if pilot tone present, do stereo demuxing
    for(int i=0; i<InLength; i++)
    {
            TYPEREAL in = m_RawFm[i];
            //Left minus Right signal is created by multiplying by 38KHz recovered pilot
            // scale by 2 since DSB amplitude is half of the Right plus Left signal
            LminusR = 2.0 * in * sin( m_PilotPhase[i]*2.0);
            pOutData[i].re = in + LminusR;          //extract left and right signals
            pOutData[i].im = in - LminusR;
    }
}
else
{      //no pilot so is mono. Just copy real FM demod into both right and left channels
        for(int i=0; i<InLength; i++)
        {
            pOutData[i].re = m_RawFm[i];
            pOutData[i].im = m_RawFm[i];
        }
 }
```

The 19kHz pilot recovery PLL code is this:

```
for(int i=0; i<InLength; i++)
{
    Sin = sin(m_PilotNcoPhase);           //Create 19Khz NCO sin/cos sample
    Cos = cos(m_PilotNcoPhase);
    //complex multiply input sample by NCO's  sin and cos
    tmp.re = Cos * pInData[i].re - Sin * pInData[i].im;
    tmp.im = Cos * pInData[i].im + Sin * pInData[i].re;
    //find current sample phase after being shifted by NCO frequency
    TYPEREAL phzerror = -arctan2(tmp.im, tmp.re);
    //create new NCO frequency term
    m_PilotNcoFreq += (m_PilotPllBeta * phzerror);        //  radians per sampletime
    //clamp NCO frequency so doesn't get out of lock range
    if(m_PilotNcoFreq > m_PilotNcoHLimit)
            m_PilotNcoFreq = m_PilotNcoHLimit;
    else if(m_PilotNcoFreq < m_PilotNcoLLimit)
            m_PilotNcoFreq = m_PilotNcoLLimit;
    //update NCO phase with new value
    m_PilotNcoPhase += (m_PilotNcoFreq + m_PilotPllAlpha * phzerror);
    m_PilotPhase[i] = m_PilotNcoPhase + m_PilotPhaseAdjust;//phase fudge for exact phase delay
    //create long average of error magnitude for lock detection
    m_PhaseErrorMagAve = (1.0-m_PhaseErrorMagAlpha)*m_PhaseErrorMagAve +
                                    m_PhaseErrorMagAlpha*phzerror*phzerror;
}
m_PilotNcoPhase = fmod(m_PilotNcoPhase, K_2PI); //keep radian counter bounded
if(m_PhaseErrorMagAve < LOCK_MAG_THRESHOLD)
    return TRUE;
else
    return FALSE;
}
```

Because of phase delay introduced by the 19kHz filters and PLL, it must be compensated for before using it to recover the L-R signal. This is done by adding a fudge factor to the phase when generating the 38kHz sinewave. The factor is dependent on sample frequency and was detemined experimentally by adjusting for best stereo separation. Its value is nearly linear with sample rate so a simple linear fit equation is used to calculate the fudge factor for any sample rate.

A few final steps are needed before sending to the soundcard.

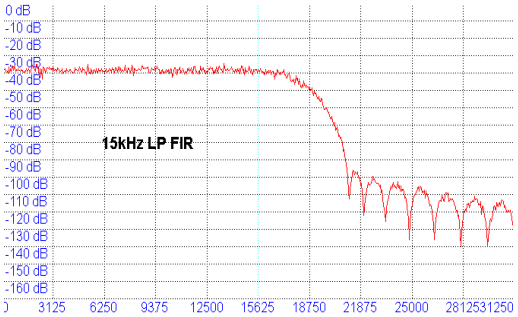


One step is to reduce the sample rate and this is done using several halfband filters to decimate in steps of 2 until close to the final soundcard rate of 48kHz.

```
//decimate by 2's down close to final audio rate
if(m_pDecBy2A)
    InLength = m_pDecBy2A->DecBy2(InLength, pOutData, pOutData);
if(m_pDecBy2B)
    InLength = m_pDecBy2B->DecBy2(InLength, pOutData, pOutData);
if(m_pDecBy2C)
    InLength = m_pDecBy2C->DecBy2(InLength, pOutData, pOutData);

m_LPFilter.ProcessFilter( InLength, pOutData, pOutData);   //rolloff audio above 15KHz
ProcessDeemphasisFilter(InLength, pOutData, pOutData);  //50 or 75uSec de-emphasis one pole filter
m_NotchFilter.ProcessFilter( InLength, pOutData, pOutData);  //notch out 19KHz pilot
```

The audio is then low pass filtered to reduce signals above 15kHz using a FIR LP filter.

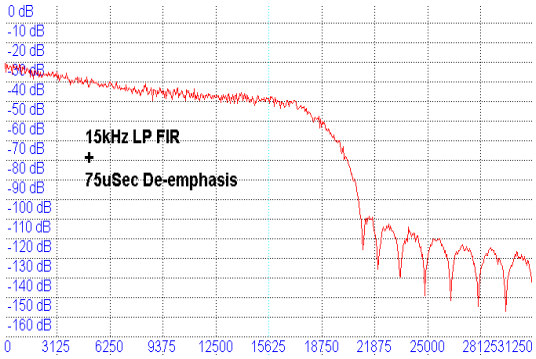



Since the FM broadcast stations use a preemphasis filter to increase the S/N performance of FM by enhancing high audio frequencies, a deemphasis filter must be performed to restore the audio spectrum. This is just a simple one pole filter specified in microseconds. In the US it is 75uSeconds and elsewhere 60uSeconds. The exponential averaging filter will do nicely here as it is simple and implements just such a filter. It is described in detail earlier in the S-Meter section.

```
m_DeemphasisAve = (1.0-m_DeemphasisAlpha)*m_DeemphasisAve + m_DeemphasisAlpha*InBuf[i];
```

where DeemphasisAlpha is $ALPHA = 1 - e^{-T/Tau}$
where T is the sample period in seconds.
The time constant of an equivalent RC filter is Tau=1/RC.

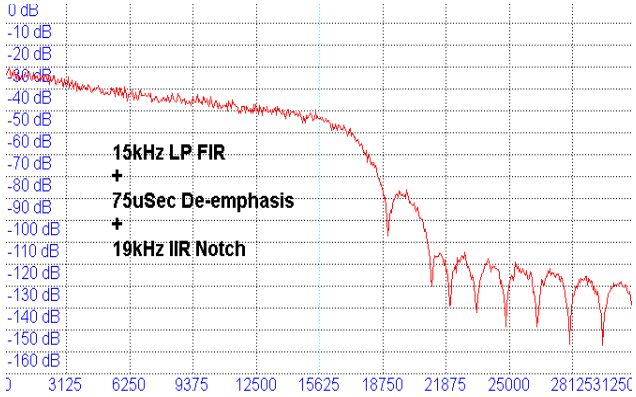
Adding the de-emphasis response is shown below.



Finally it may be a good idea to remove the 19kHz pilot tone from the audio. Most cannot hear it but it may irritate your dog or overload your amplifier.

An IIR notch filter with a Q of 5 is used.

The final response of all these post filters is this:



## *4.16. RDS Encoding/Decoding*

The Radio Data System( RDS) signal is an optional digital signal that is a double sideband modulated signal at 57kHz within the FM demodulated spectrum. The RDS receiver is implemented in the same CWFmDemod Class but will be discussed separately in this section.

The RDS specification can be found on the Internet.

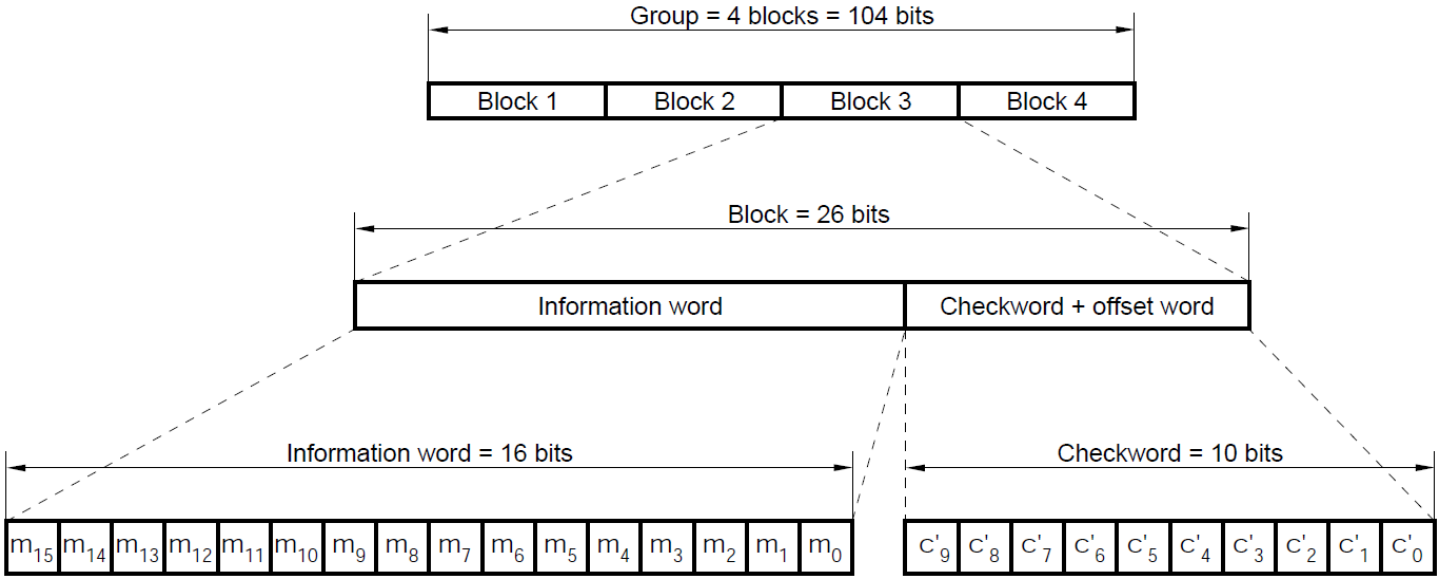
http://www.nrscstandards.org/

http://www.rds.org.uk/2010/RDS-Specification.htm

There are two variants, the RDS standard and the Radio Broadcast Data System(RBDS) which is found in North America. Since the low level signaling is the same for both systems, the differences will not be dealt with since they are at the higher level decoding of the information.

### 4.16.1 RDS Signal Encoding

The the method of generating an RDS signal will be discussed first in order to better understand the decoding process. The RDS specification gives a complete description of the format of the digital encoding as well as the analog modulation process. For this discussion we will start at the upper level where information bits are formatted into a “Group” consisting of 4 blocks of 26 bits. The format of the actual message data is all described in the specification so will not be described.




The Group of 104 bits is repeated and transmitted continuously most significant(leftmost) bit first. Each of the four 26 bit blocks consist of 16 bits of payload data and a 10 bit check word. The check word for each block has a unique offset word added(XOR'ed in the error correction world) to it in order to distinguish between the different data blocks. To make it more complicated, there is a fifth data block that can be used in place of the third block that has its own offset word and is called a C' block. The spec seems to jump around between calling the blocks 1 to 4 and A to D.

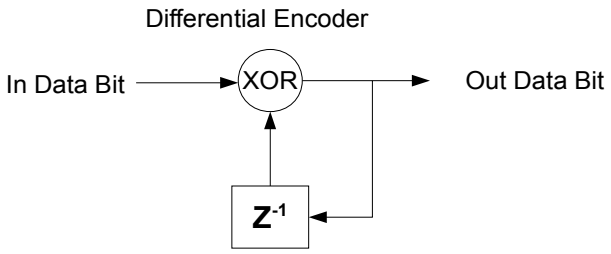
The check word can be created directly from the generator polynomial by getting the remainder of the division of the message word by the block code polynomial, or use a generator matrix and a look-up table created from the information in the specification annex B and a little code.

```
//Generator matrix from RDS spec for creating check sum word
const quint32 CHKWORDGEN[16] =
{
        0x077,  //00 0111 0111
        0x2E7,  //10 1110 0111
        0x3AF,  //11 1010 1111
        0x30B,  //11 0000 1011
        0x359,  //11 0101 1001
        0x370,  //11 0111 0000
        0x1B8,  //01 1011 1000
        0x0DC,  //00 1101 1100
        0x06E,  //00 0110 1110
        0x037,  //00 0011 0111
        0x2C7,  //10 1100 0111
        0x3BF,  //11 1011 1111
        0x303,  //11 0000 0011
        0x35D,  //11 0101 1101
        0x372,  //11 0111 0010
        0x1B9   //01 1011 1001
};

//////////////////////////////////////////////////////////////////////////////
//      Create data blk with chkword from 16 bit 'Data' with 'BlockOffset'.
// Returns 26 bit block with check word.
//////////////////////////////////////////////////////////////////////////////
quint32 CWFmMod::CreateBlockWithCheckword(quint16 Data, quint32 BlockOffset)
{
quint32 block = (quint32)Data<<10;      //put 16 msg data bits into block
        for(int i=0; i<NUMBITS_MSG; i++)
        {   //do matrix operation on data bits 15 to 0
            //Since generator matrix is in systematic form
            //(first 16 columns are a diagonal identity matrix),
            //just XOR from table where message bit is a one.
            if(Data & 0x8000)       //if msg bit 15 is 1, XOR with generator matrix value
                    block = block ^ CHKWORDGEN[i];
            Data <<= 1;    //go to next bit position
        }
        block = block ^ BlockOffset;    //add in block offset word
        return block;
}
```

Now that we have all the message ones and zeros needed to shift out, they need to be differentially encoded by XOR'ing the previous bit with the current one. This is done so that the absolute polarity of the bit is not important in cases where the modulator or demodulator may invert the bits. The penalty for this is that an error in one bit will cause an error in the next bit as well.

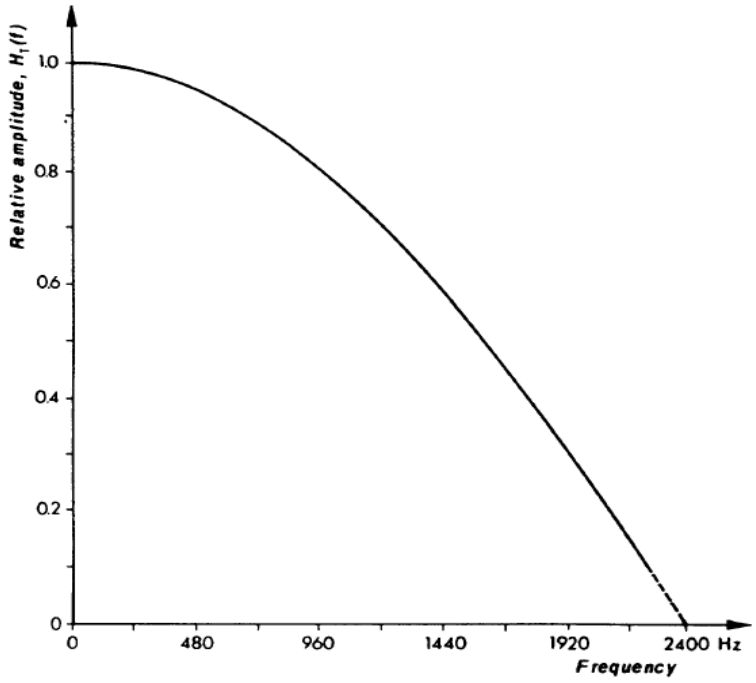
```
//differential encode output bit by XOR with previous output bit
//return +1.0 for a '1' and -1.0 for a '0'
if( m_RdsLastBit ^ bit )
{
    m_RdsLastBit = 1;
    return 1.0;
}
else
{
    m_RdsLastBit = 0;
    return -1.0;
}
```



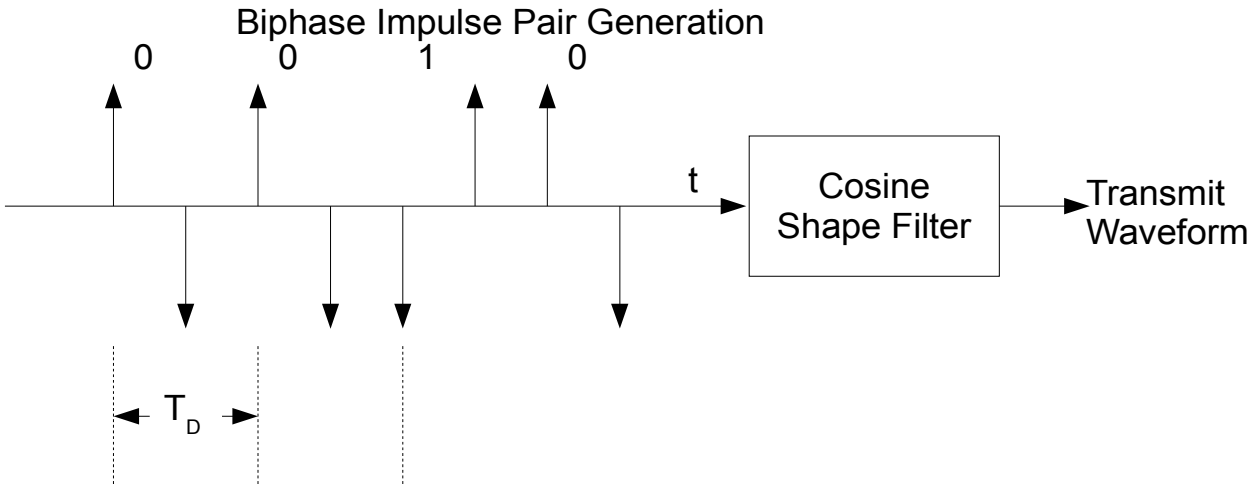

The next step is to create the actual modulation waveform from the string of bits. From the RDS specification, the following pulse shape is given as the bit shaping function.

$$H_T(f) = \begin{cases} \cos\dfrac{\pi f t_d}{4} & \text{if } 0 \le f \le 2/t_d \\ 0 & \text{if } f > 2/t_d \end{cases}$$



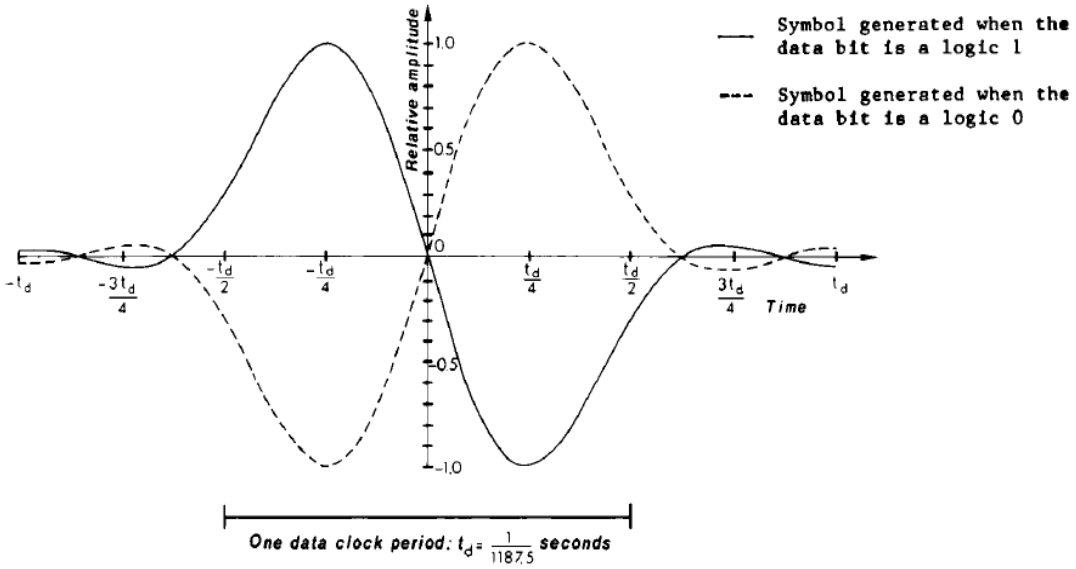

To create this waveform you could create 2 impulse pair functions corresponding to the one or zero data and run them through a cosine shaping filter to get the final output waveform.




If one were to implement the shaping filter with an FIR filter, then the impulse response of the cosine shaping filter needs to be found.

The RDS specification shows a plot of the time domain function of the one or zero pulse pair, but alas does not give any equations to generate it.




After some searching, a paper was found from a course at the authors alma mater( University of Illinois ECE 463) that appears to give an approximation to the above pulse response.

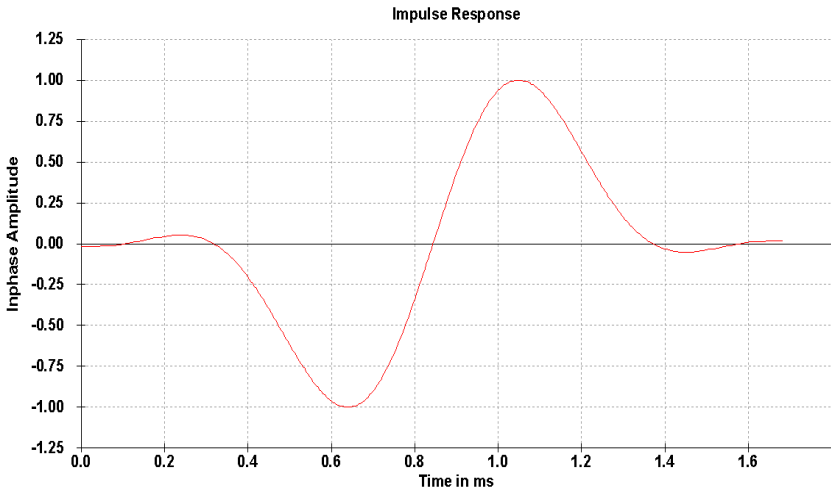
http://courses.engr.illinois.edu/ece463/Projects/RBDS/RBDS_project.doc

$$u(t) = \frac{\pm 3}{4}\cos(4\pi x)\left[\frac{1}{\left(\frac{1}{x} - 64x\right)} - \frac{1}{\left(\frac{9}{x} - 64x\right)}\right]$$

where $x = \frac{t}{T_d}$ $\quad T_d = data\ period$

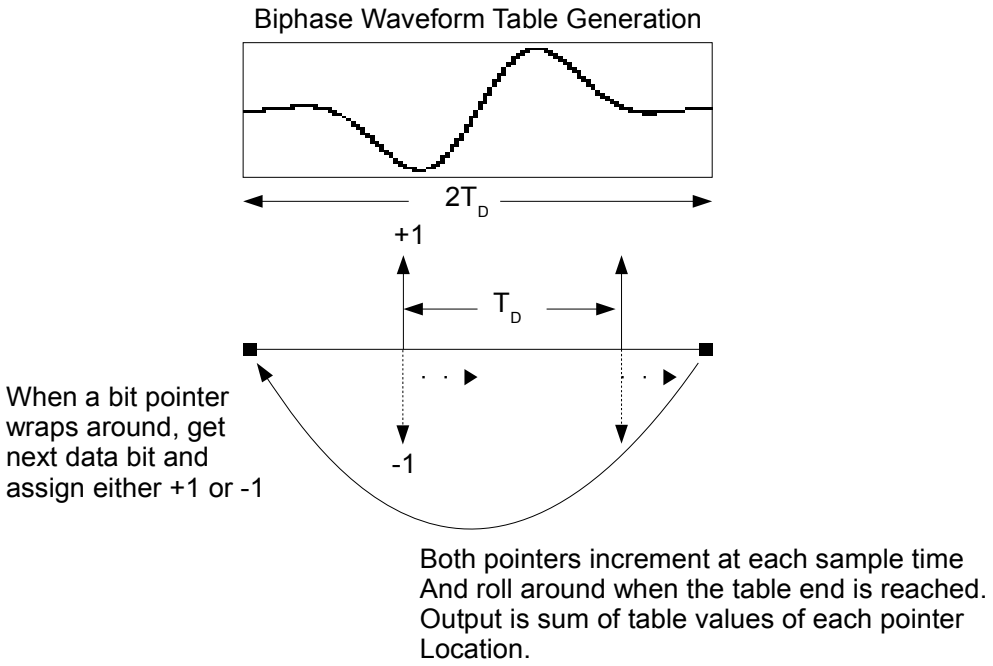
Plotting this function shows that it matches the one in the specification fairly well.



To generate a zero or one pulse pair, just output the above function or the negative of it directly. Note that the function is twice the length of a single bit so there is overlap between bits. Two pointers continually traversing this function with one lagging by one bit time can be used to produce the desired waveform.

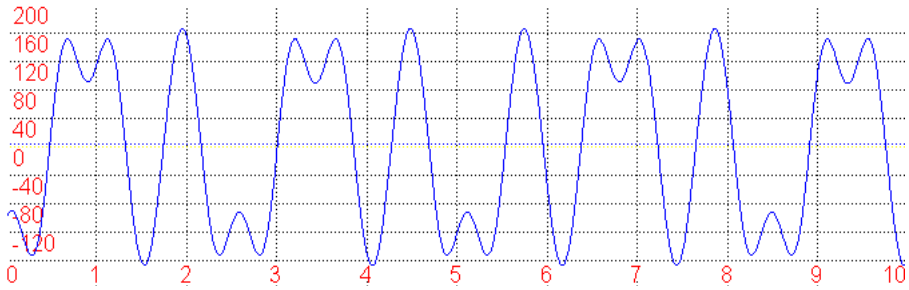


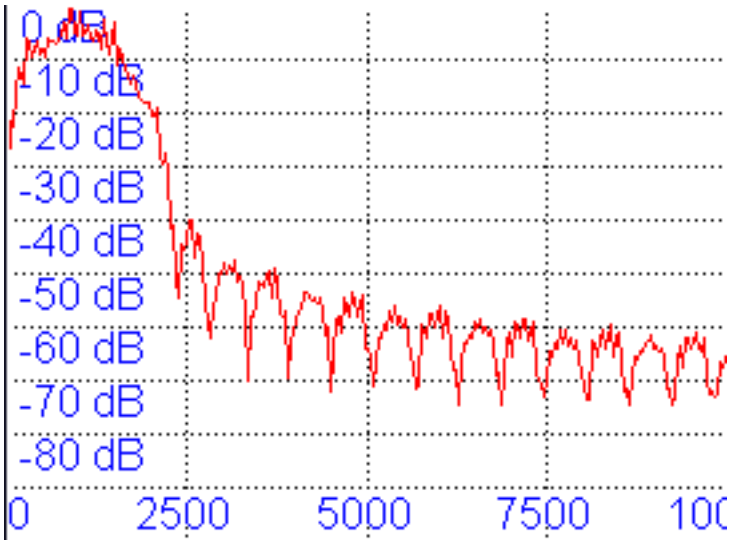
The following routine implements this table look-up method of RDS signal generation in the CWFmMod class.

```
void CWFmMod::CreateRdsSamples(int InLength , TYPEREAL* pBuf)
{
int n1;
int n2;
double rdsperiod = 1.0/RDS_BITRATE;
double rds2period = 2.0/RDS_BITRATE;
        for(int i= 0; i<InLength; i++)
        {
                n1 = (int)(m_RdsTime * m_SampleRate);   //create integer index
                //calculate index positions of both pointers
                if(m_RdsTime > rdsperiod)
                        n2 = (int)((m_RdsTime-rdsperiod) * m_SampleRate);
                else
                        n2 = (int)( (m_RdsTime+rdsperiod) * m_SampleRate);
                //if a pointer wraps to zero, get next new data bit value
                if(0==n1)
                        m_RdsD1 = CreateNextRdsBit();
                if(0==n2)
                        m_RdsD2 = CreateNextRdsBit();
                //get both table values and add together for output sample value
                pBuf[i] = m_RdsD1*m_RdsPulseCoef[n1] + m_RdsD2*m_RdsPulseCoef[n2];
                //manage running floating point time position
                m_RdsTime += m_RdsSamplePeriod;
                if(m_RdsTime >= rds2period)
                        m_RdsTime -= rds2period;
        }
}
```

The following is a plot of the time domain output from this RDS generating routine.




The measured spectrum plot looks like this.

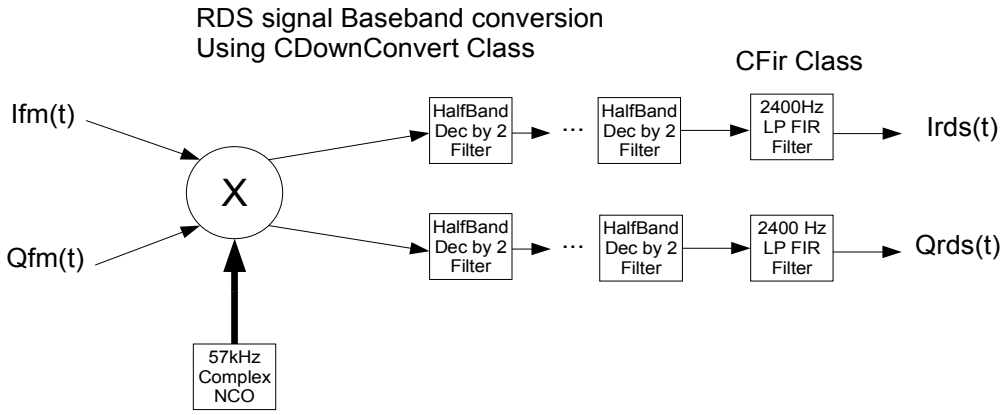


### 4.16.2 RDS Signal Demodulation

The process of demodulating and decoding the RDS signal from the FM demodulated data stream is done by first shifting the 57KHz centered RDS DSB signal to base band at 0Hz. Since we have already created a complex signal in the stereo demodulation process, one can use the existing CDownConvert class to both translate to base band and also to decimate the sample rate down to a rate closer to the RDS signal bandwidth. A reasonable final rate is around 10 to 20Ksps.

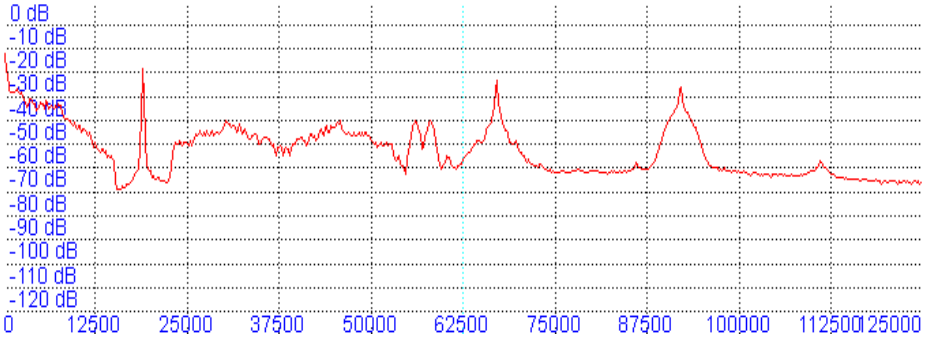
A FIR LP filter after the down conversion is used to isolate the RDS signal.

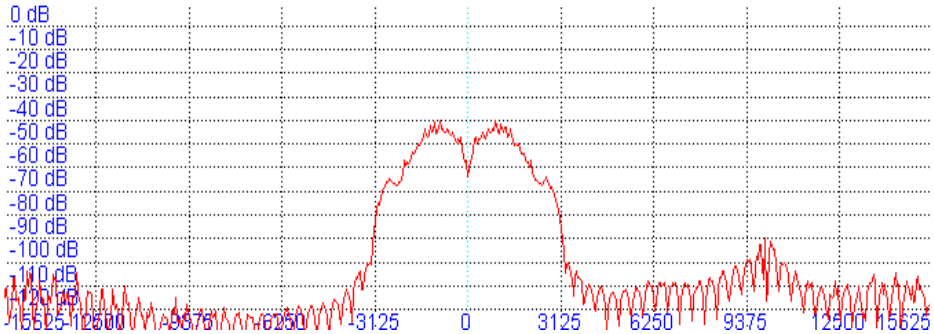



The following code segment from the CWFmDemod class performs the operation.

```
//translate 57KHz RDS signal to base band and decimate RDS complex signal
int length = m_RdsDownConvert.ProcessData(lnLength, m_CpxRawFm, m_RdsRaw);
//filter base band RDS signal
m_RdsBPFilter.ProcessFilter(length, m_RdsRaw, m_RdsRaw);
```

Here is an off the air signal before and after performing the above down conversion/filtering process:

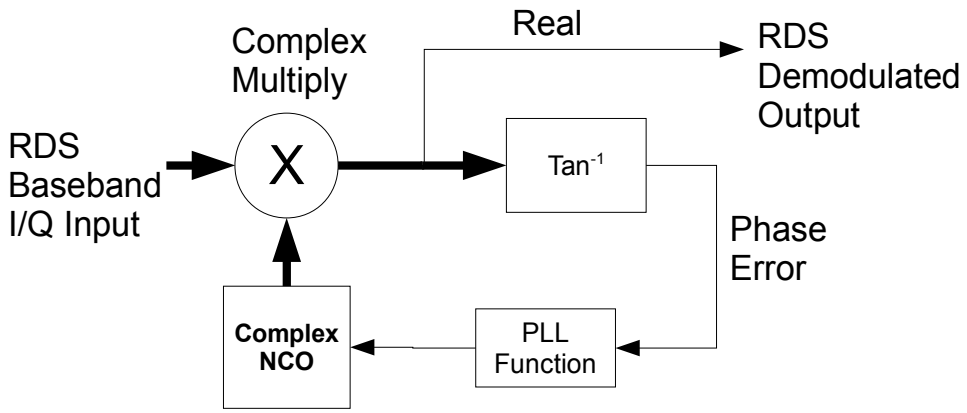


The first thought would be to use the 19kHz pilot tone to create the 57kHz NCO and all would be nice and phase locked and ready to extract the RDS DSB signal. Unfortunately even though the spec indicates that should be the case, many stations do not phase lock the RDS signal to the 19kHz stereo pilot signal and even if they did, the specification clearly states the RDS signal can be used in mono mode where there isn't a 19kHz pilot. This means we must demodulate the DSB signal as if it were independent of any reference.

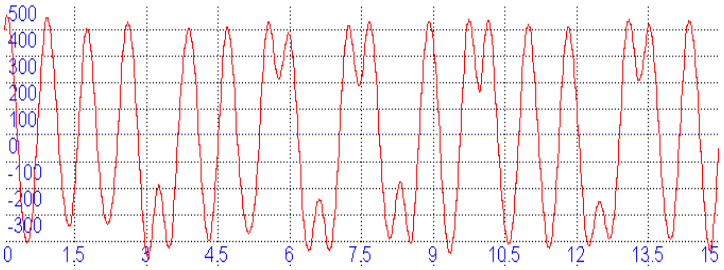
Stealing from the synchronous AM detector, we can use that method to lock a PLL to the base band DSB signal and recover the data.




The following code segment implements the RDS DSB PLL demodulation.

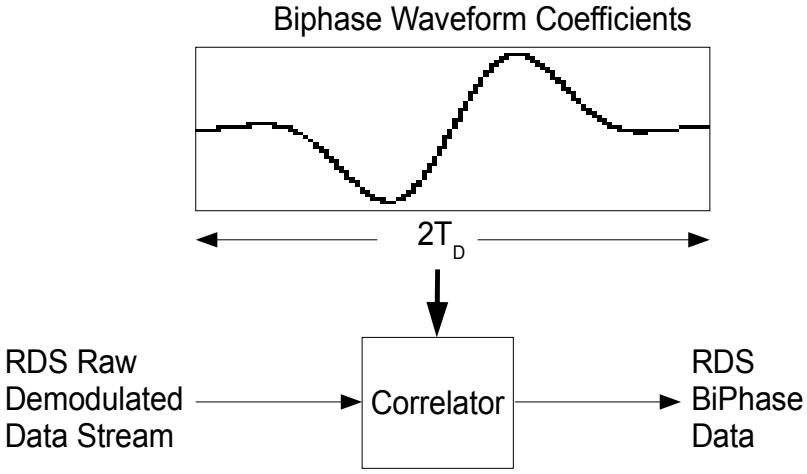
```
for(int i=0; i<InLength; i++)
{
   Sin = sin(m_RdsNcoPhase);
  Cos = cos(m_RdsNcoPhase);
  //complex multiply input sample by NCO's  sin and cos
  tmp.re = Cos * pInData[i].re - Sin * pInData[i].im;
  tmp.im = Cos * pInData[i].im + Sin * pInData[i].re;
  //find current sample phase after being shifted by NCO frequency
  TYPEREAL phzerror = -arctan2(tmp.im, tmp.re);
  //create new NCO frequency term
  m_RdsNcoFreq += (m_RdsPllBeta * phzerror);   //  radians per sampletime
  //clamp NCO frequency so doesn't get out of lock range
  if(m_RdsNcoFreq > m_RdsNcoHLimit)
        m_RdsNcoFreq = m_RdsNcoHLimit;
  else if(m_RdsNcoFreq < m_RdsNcoLLimit)
        m_RdsNcoFreq = m_RdsNcoLLimit;
  //update NCO phase with new value
  m_RdsNcoPhase += (m_RdsNcoFreq + m_RdsPllAlpha * phzerror);
  pOutData[i] = tmp.im;
}
m_RdsNcoPhase = fmod(m_RdsNcoPhase, K_2PI);     //keep radian counter bounded
```

A plot of the output of the above PLL from an off the air signal looks very much like the RDS signal that was created earlier.




Now the fun begins. What is a one and what is a zero in the above waveform? Remember that the original encoding is a pulse pair of +1 then -1 or -1 then +1, then filtered.

If one were to perform a correlation of this demodulated signal with the time domain function used earlier to create the output signal, we would have implemented a matched filter that would give a maximum amplitude when the incoming signal was closest to looking like the generated pulse shape.

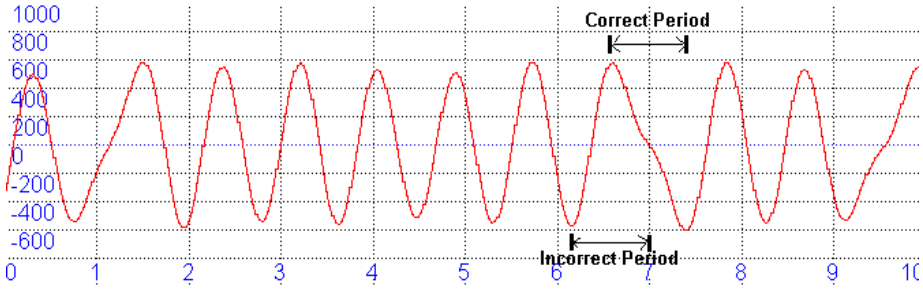



The correlator is just an FIR filter whose coefficients are the BiPhase waveform of the RDS data so we can use the normal CFir class to implement. The FIR coefficients are calculated exactly like was done in the RDS signal generation discussion.

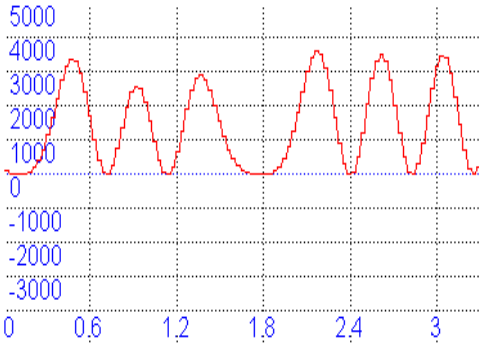
```
//run matched filter correlator to extract the bi-phase data bits
m_RdsMatchedFilter.ProcessFilter(length,m_RdsMag,m_RdsData);
```

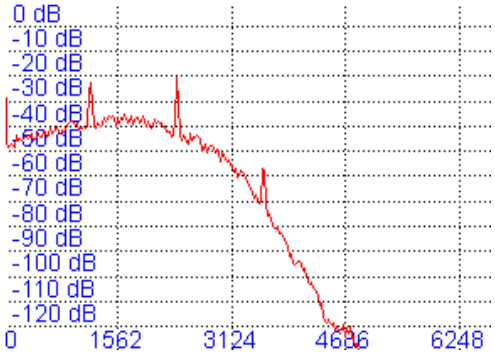
The output of this matched filter appears as follows. Now a 'One' is a positive peak and a 'Zero' is a negative peak.

Nothing is easy. As is shown, there is an ambiguity as to where the bit center begins. If one picks the incorrect phase, then you would be sampling some points when they are near zero. The correct sample phase is where there is always a peak at the sample time.

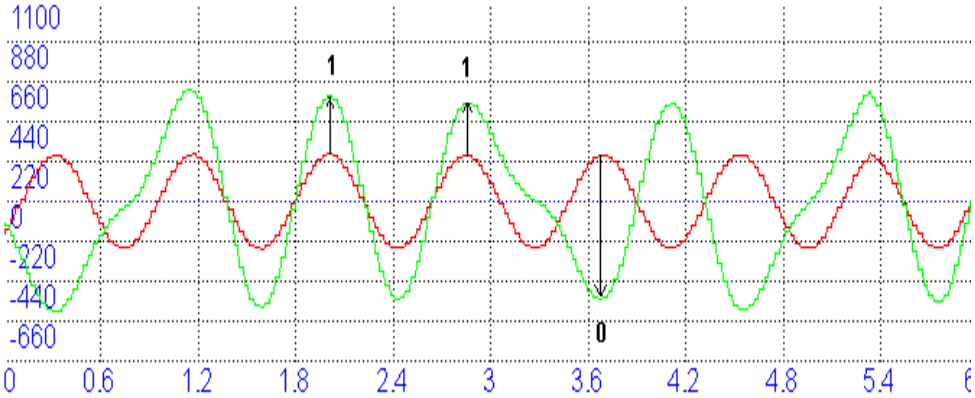


Note there are two frequency components in the biphase data signal. There is a component at ½ the data rate and also one at the data rate. By squaring the data signal we can get peaks at the data rate and twice the data rate. Using the component that is at the data rate will provide a bit sampling clock with the correct phase. Below is a plot of the squared data and its spectrum. Note the components at the data rate and twice the data rate.




An IIR bandpass filter centered at the data rate, will extract a signal that is just at the data rate and will have its positive peaks occur exactly at the correct sample point in the data. An IIR bandpass filter with a very high Q will behave almost like an oscillator and will ring at the desired frequency and phase of the incoming signal. Below is a plot showing the data signal and the output of the IIR sample rate filter. Note that the positive peaks of the filter output correspond to the correct sample point in the data stream.

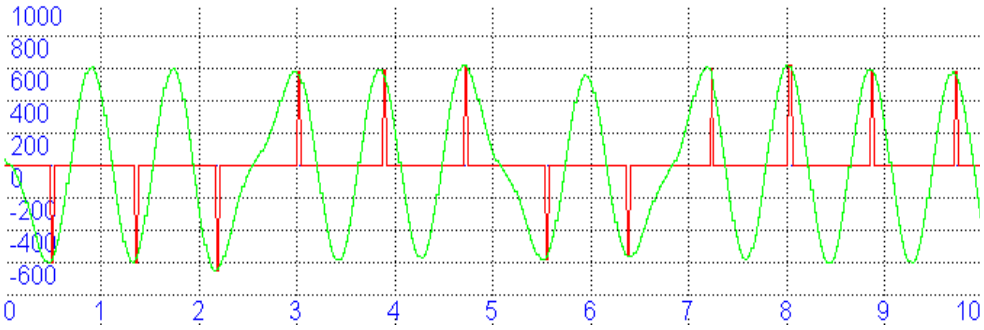



The positive peak is found by looking at the slope of the sine wave and when the slope changes from positive to negative, that is the correct sample time to determine the data bit. If the data is positive at the sample time then is is a '1', if negative then a '0'.

The following code segment implements this logic and extracts the data bit for further processing.

```
//create bit sync signal in m_RdsMag[] by squaring data
for(int i=0; i<length; i++)
   m_RdsMag[i] = m_RdsData[i]* m_RdsData[i];   //has high energy at the bit clock rate and 2x bit rate
//run Hi-Q resonator filter that create a sin wave that will lock to BitRate clock and not 2X rate
m_RdsBitSyncFilter.ProcessFilter(length, m_RdsMag, m_RdsMag);

//now loop through samples to determine where bit position is and extract binary digital data
for(int i=0; i<length; i++)
{
   TYPEREAL Data = m_RdsData[i];
   TYPEREAL SyncVal = m_RdsMag[i];
   //the best bit sync position is at the positive peak of the sync sine wave
  TYPEREAL Slope = SyncVal - m_RdsLastSync;       //current slope
  m_RdsLastSync = SyncVal;
  //see if at the top of the sine wave
  if( (Slope<0.0) && (m_RdsLastSyncSlope*Slope)<0.0 )
  { //are at sample time so read previous bit time since
     // we are one sample behind in sync position
          int bit;
       if(m_RdsLastData>=0)
                bit = 1;
          else
                bit = 0;
          //need to XOR with previous bit to get actual data bit value
          ProcessNewRdsBit(bit^m_RdsLastBit);       //go process new RDS Bit
          m_RdsLastBit = bit;
   }
```

The following plot shows the biphase data signal and the extracted data bit at each sample time.



### 4.16.3 RDS Signal Decoding

After all that we finally have actual ones and zeros to work with. The next step is to figure out where the beginning of a data group is. Unfortunately there is no sync pattern to determine the block boundaries. The only way is to assume we are at the start of a block and then calculate the check sum and if it is OK then we were at the actual block boundary. If not then try again at the next bit time. Once the block boundaries are found then the bits can be shifted in and processed at the end of each block and not every bit time.

#### 4.16.3.a Syndrome Decoding

A syndrome is a polynomial result that can be used to detect and/or correct errors in certain block codes. The RDS coding uses this to find and try to detect some errors. In the RDS specification, a block decoding scheme is described using shift registers and logic as would be used for a hardware implementation. The same scheme can be used but optimized a little for a software implementation.

Several constants and tables can be used to make the implementation easier. First the 'Parity Check Matrix H' is given in the specification and can be used directly to calculate the syndrome of a block. Similar to using the “Generator Matrix G” to create data blocks, a matrix operation can be performed that will return the syndrome of a received data block.

$$H = \begin{vmatrix} 1000000000 \\ 0100000000 \\ 0010000000 \\ 0001000000 \\ 0000100000 \\ 0000010000 \\ 0000001000 \\ 0000000100 \\ 0000000010 \\ 0000000001 \\ 1011011100 \\ 0101101110 \\ 0010110111 \\ 1010000111 \\ 1110011111 \\ 1100010011 \\ 1101010101 \\ 1101110110 \\ 0110111011 \\ 1000000001 \\ 1111011100 \\ 0111101110 \\ 0011110111 \\ 1010100111 \\ 1110001111 \\ 1100011011 \end{vmatrix}$$

This has been precalculated and includes the modifications needed since the RDS code is a shortened subset of the original cyclic code that has a 341 bit length. The syndrome is just the matrix multiply of the Parity Check matrix H and the received 26 bit codeword. Since the first ten rows are an identity matrix, it just copies the first 10 bits of the block directly into the syndrome. The remaining 16 bits are then processed by XOR'ing the syndrome with corresponding table entry if the corresponding data block bit is a one.

The following code segment calculates the syndrome of m_InBitStream. Similar to the block offset addition when creating blocks, a unique Syndrome Offset is added depending on which block we are looking for.

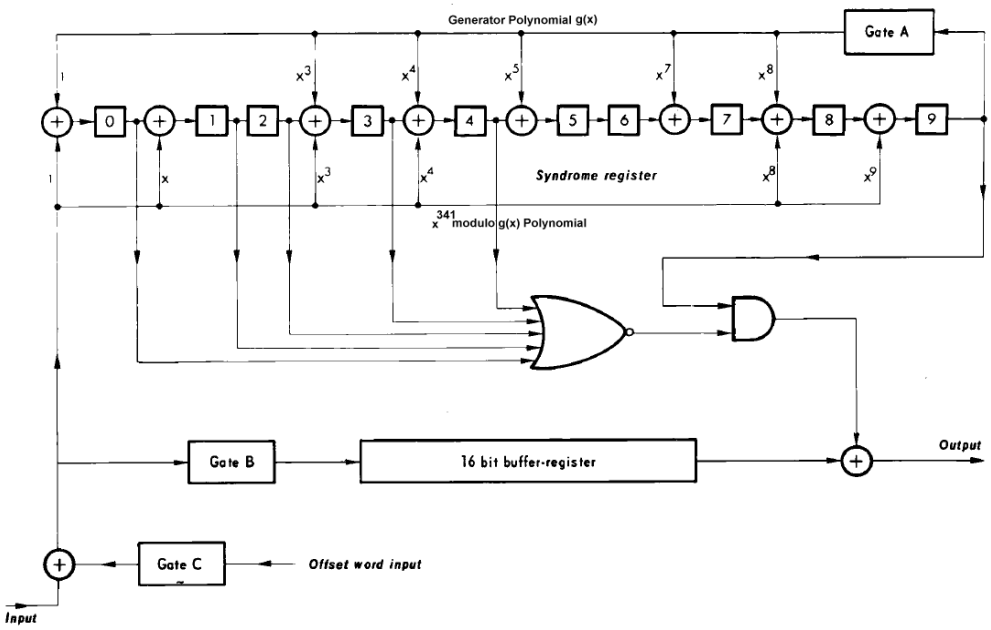
```
quint32 testblock = (0x3FFFFF & m_InBitStream); //isolate bottom 26 bits
//copy top 10 bits of block into 10 syndrome bits since first 10 rows
//of the check matrix is just an identity matrix(diagonal one's)
quint32 syndrome = testblock>>16;
for(int i=0; i<NUMBITS_MSG; i++)
{       //do the 16 remaining bits of the check matrix multiply
   if(testblock&0x8000)
           syndrome ^= PARCKH[i];
   testblock <<= 1;
}
syndrome ^= SyndromeOffset;             //add depending on desired block
```

### 4.16.3.b Error Correction

If the syndrome is not zero then there are errors. One could stop here and just use the syndrome as a simple good/bad block finder. However, one can use the properties of linear block codes to try and correct certain types of errors and improve RDS reception in noisy conditions.

Again in the RDS specification a block diagram of a hardware implementation of a decoder that can correct up to 5 consecutive(burst) errors. A form of Meggitt decoder is implemented that also includes the logic to generate the syndrome as well.




For the software implementation, we have already pre-calculated the syndrome taking into account the block offset and $x^{241}$modulo g(x) factor so all that is left is the shifting out of the message bits as we see if any can be corrected. The algorithm is that if the msb of the syndrome is a one and the bottom five bits of the syndrome are zero, then the corresponding message bit being shifted out can be corrected by inverting it. The syndrome is then shifted and the next msb examined etc.

The following code segment implements this

```
quint32 correctedbits = 0;
quint32 correctmask = (1<<(NUMBITS_BLOCK-1));   //start pointing to msg msb
//Run Meggitt FEC algorithm to correct up to 5 consecutive burst errors
for(int i=0; i<NUMBITS_MSG; i++)
{
   if(syndrome & 0x200)    //chk msbit of syndrome for error state
   {       //is possible bit error at current position
           if(0 == (syndrome & 0x1F) ) //bottom 5 bits == 0 tell it is correctable
           {       // Correct i-th bit
                   m_InBitStream ^= correctmask;
                   correctedbits++;
                   syndrome <<= 1;         //shift syndrome to next msb
           }
           else
           {
                   syndrome <<= 1; //shift syndrome to next msb
                   syndrome ^= CRC_POLY;   //recalculate new syndrome if bottom 5 bits not zero
           }                                                       //and syndrome msb bit was a one
   }
   else
   {       //no error at this bit position so just shift to next position
           syndrome <<= 1; //shift syndrome to next msb
   }
   correctmask >>= 1;      //advance correctable bit position
```

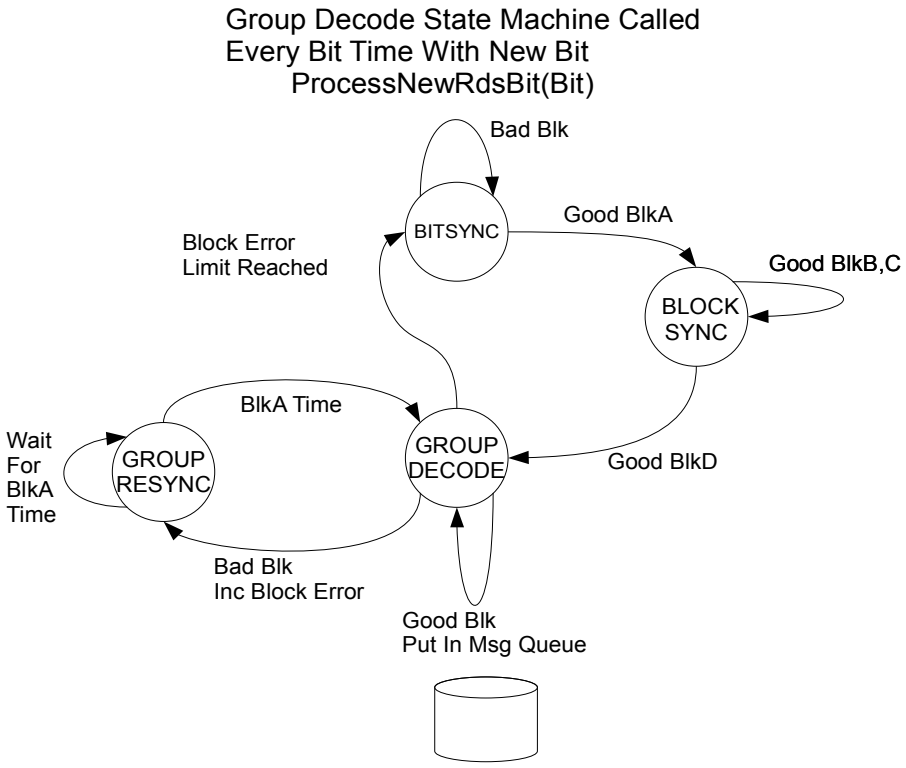
```
    }
    syndrome &= 0x3FF;      //isolate syndrome bits if non-zero then still an error
```

This is all implemented in the CheckBlock(quint32 SyndromeOffset, int UseFec) routine. It can be called with a Boolean to not use the FEC section. This is useful during initial synchronization to insure a good block.

One problem with the small block size that the RDS uses is that in random noise, false good blocks can be indicated. Enabling the FEC increases the probability of false blocks since it ends up correcting the noisy bits and makes 'good' block out of them. False blocks have been observed at over once per second in random noise.

One way around this is to initially require that the four blocks in a group arrive exactly in order ABCD without any FEC. The probability of noise generating this exact sequence is much lower and gives a high confidence that the correct bit position in the data stream is found. Another way might be to require a majority of Block A's to be equal before saying we have the correct bit position.

In CuteSDR, a state machine is implemented that manages all the synchronization and error limiting. This routine could be improved or modified to measure bit error rates and take advantage of redundant source blocks.




State BITSYNC checks for a good block A at every bit time position.

State BLOCKSYNC checks at every block time for good sequential blocks B,C and D.

State GROUPDECODE continuously checks blocks A-D and places into the data queue. If a bad block is received, it jumps to the GROUPRESYNC state which just waits until a new block A is received then it goes back to the GROUPDECODE state. If a bad block error limit is reached, it goes back to the BITSYNC state to start over.

Good Data Groups of the four blocks are placed in a queue as they are decoded. The queue data structure is as follows. The upper level GUI code can then remove groups from the queue at its leisure and decode into whatever user format is desired.

```
typedef struct _RDS_GRPS
{
        quint16 BlockA;
        quint16 BlockB;
        quint16 BlockC;
        quint16 BlockD;
}tRDS_GROUPS;
```

The upper level GUI code can retrieve RDS data blocks by calling the access routine “GetNextRdsGroupData()” with a pointer to a tRDS_GROUP structure to hold the data. It returns a zero if the queue is empty or if the data group is exactly the same as the previous one. An all zero group is stuffed into the queue whenever RDS sync is lost to indicate that the GUI should restart the upper level decode since its possible the station has been changed or the signal is lost.

It should be noted that the RDS signal strength varies from station to station and is not totally dependent on the RF signal strength. The threshold of the RDS detection is lost way before the audio becomes unreadable. Also FM multipath can degrade the RDS signal even if the audio is still readable.

The upper level GUI of CuteSDR only decodes a few of the possible fields present in the RDS groups. The PI code is decoded for US stations into a call sign except for some stations that have moved from the original RBDS specification. The program type code is decoded and also the teletext message is displayed.

## *4.17. Fractional Resampler (CFractResampler Class)*

One problem exists in that the sample rate from the radio is not the same as what is normally required by a sound card. Most sound cards operate on fixed sample rates or integer multiples of one such as 44.1Ksps or 48Ksps.

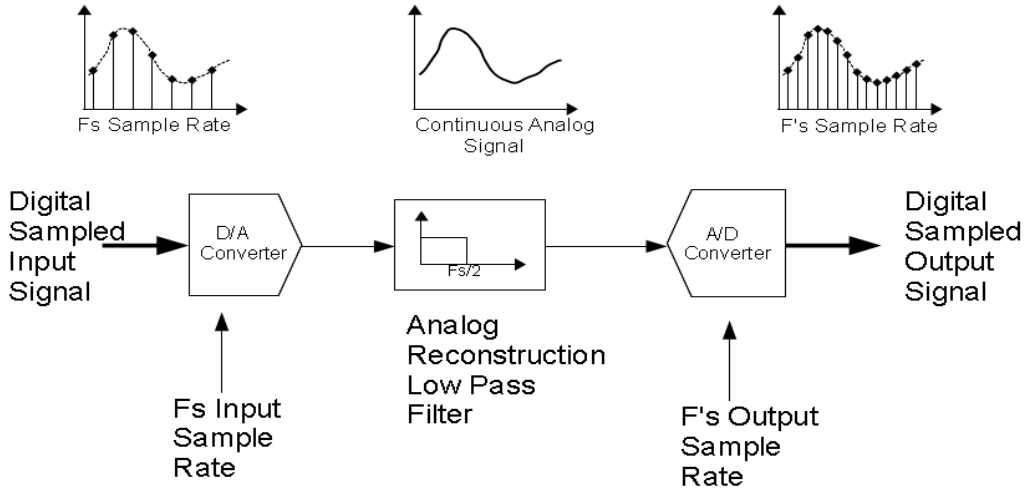
Even if the data rate from the radio after decimation was the same, there would still be a slight difference in rates since the radio and sound card clock sources are different and can be off by whatever frequency tolerance exists between the two clock sources. Some sort of fractional sample rate conversion is required.

One of the design decisions in CuteSDR was where to place the resampler. If it was placed right after the decimation stages then all the filtering and demodulation could be done at one or two sample rates simplifying their design. However, the fractional resampler's dynamic range and spurious response is not at the same level as the main filtering and could degrade performance if placed before the filters and demodulation. It was decided to place it just after the demodulation stages and before the sound card output so that it would be acting on audio data which could tolerate less dynamic range since it was after the AGC and also any spurious artifacts would be out of the range of hearing or less noticeable to the ear.

### 4.17.1 Design

There are several methods available such as interpolating up then decimating back down giving a rate change equal to the ratio of the interpolation and decimation values. This works if the rate difference is a nice rational fraction but in general this is not the case.

If one were to tackle this problem using hardware, the following method could be used.




Fs to F's Sample Rate Conversion

One could use a D/A converter and create an analog signal using the input sample rate. Then an analog low pass "reconstruction" filter can be used to generate a continuous analog copy of the input signal. This filter is required to remove all frequency components above the Nyquist frequency and if the filter were ideal, would create an exact continuous time domain copy of the original digitized signal.

Now that we have a continuous analog signal, one can now sample it at any rate using an A/D converter running at our desired new output sample rate. Sampling it at any higher rate is no problem since it is already bandwidth limited. There is a constraint if we wish to down sample in that the original input signal would still have to be bandwidth limited at the new lower rate.

A physical example of this would be to output an audio signal from a soundcard and take the analog output and feed it into the input of another soundcard that is operating of a completely different clock and at a different sample rate. The receiving soundcard data would not be dependent on the other soundcard sample rate.

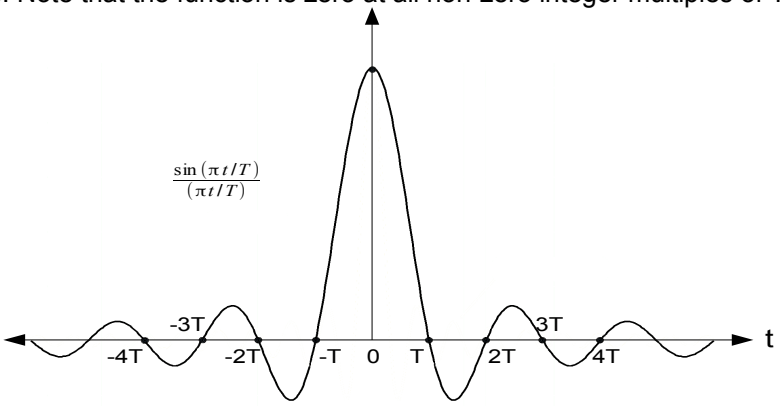
It would be nice if one could perform the same functions without resorting to hardware devices and fortunately this same idea can be implemented without leaving the digital domain.

Shannon's interpolation formula gives a mathematical means to reconstruct a bandwidth limited sampled signal.

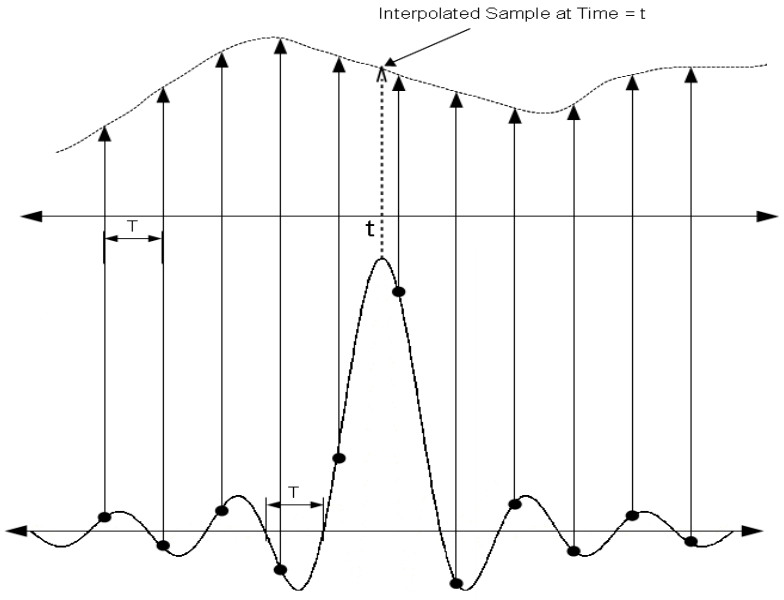
$$x(t)=\sum_{n=-\infty}^{\infty} x(nT)\,sinc\left(\frac{t-nT}{T}\right)$$

where x(t) is the continuous time domain function of our sampled input signal x[nT] and T is the sample period.

This sinc function is just a time shifted version of the ideal low pass filter which makes sense since that is what we used in our analog example. Note that the function is zero at all non-zero integer multiples of T, the sample period.

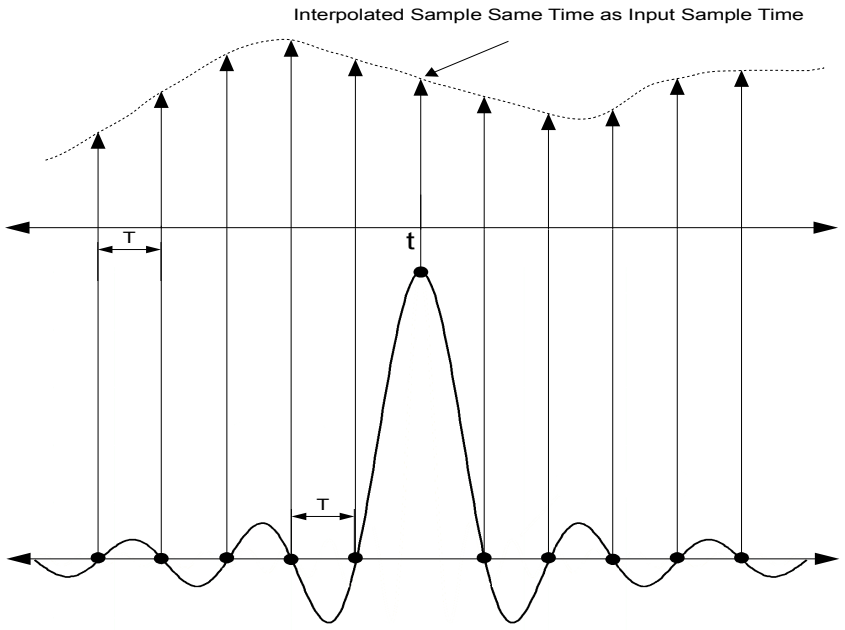



This interpolation equation means that to get the sample value at any time t, one takes the sum of all the integer sample values from +-∞ times the sinc function values at each sample time shifted by t. As shown below, one lines up the center peak of the sinc function with the desired time value and then multiplies each input sample by the corresponding sinc value.



As a little sanity check note that if the desired sample time falls exactly at an input sample time, all the sinc values are zero except for the center point so the only sinc point that is used is the center which is multiplied by the input sample giving us the expected result of just using the original input sample value.




There are several issues in implementing this interpolation scheme. The function is defined over infinity making it unrealizable. We can minimize this issue just as we do with FIR filter design by windowing the Sinc function over a finite number of samples.

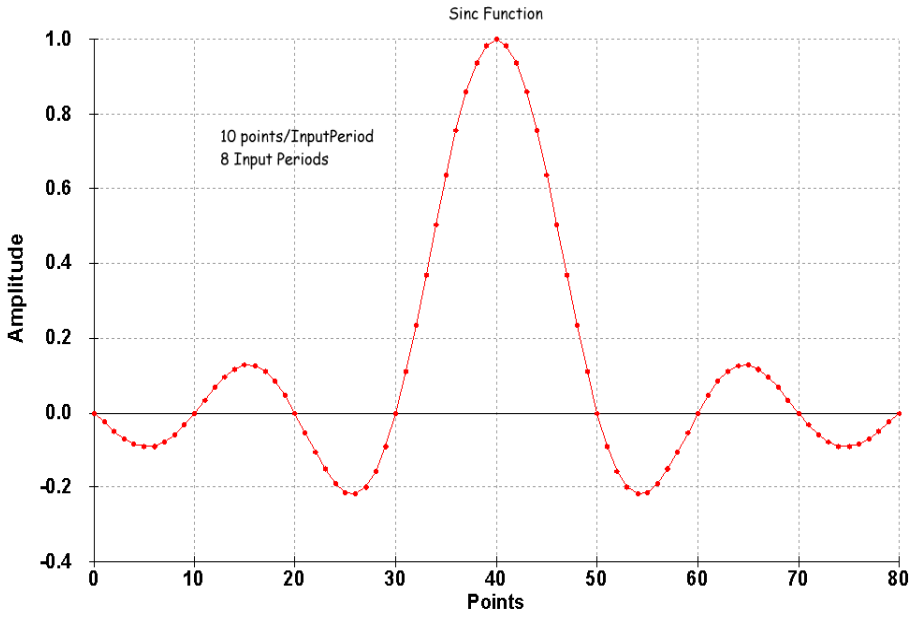
The more difficult issue is that the Sinc function is continuous and would have to be calculated at each new interpolated sample point. This could be done but is very CPU intensive since it is in the inner loop of the algorithm.

One method is to store the Sinc function values in a table. This still limits the accuracy by the number of points in the table. Good results can be obtained by using a small table and then interpolating between those points. Since PC memory is cheap, a more satisfactory solution is to just use a huge table without interpolation.

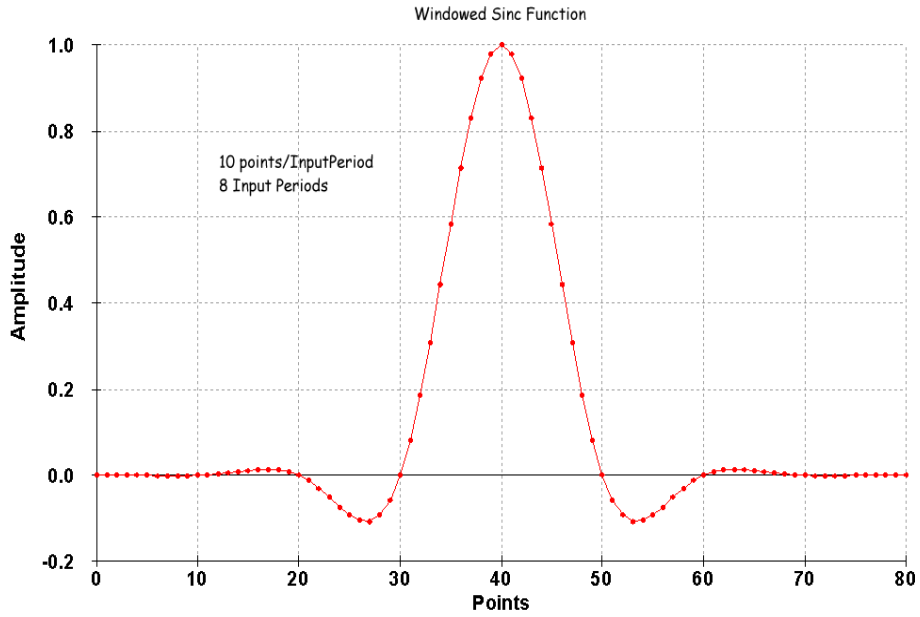
There are two main parameters used to adjust the performance of the resampler. One is the number of input samples to process per output sample which is essentially the number of “zero crossings” of the Sinc function to use. The other is the number of points stored in the table for the Sinc function.

Here is an example Sinc table showing the period and number of points relationship. This plot is before applying a window function.




After windowing the same function looks as follows.




In practice many more points are needed to provide good performance. The number of periods affects the image free bandwidth of the resampler. The more periods or “FIR filter taps”, the more useable bandwidth is available. The number of points in between periods affects the spurious performance or noise floor of the resampler since the more points the closer the table gets to the ideal continuous Sinc function.

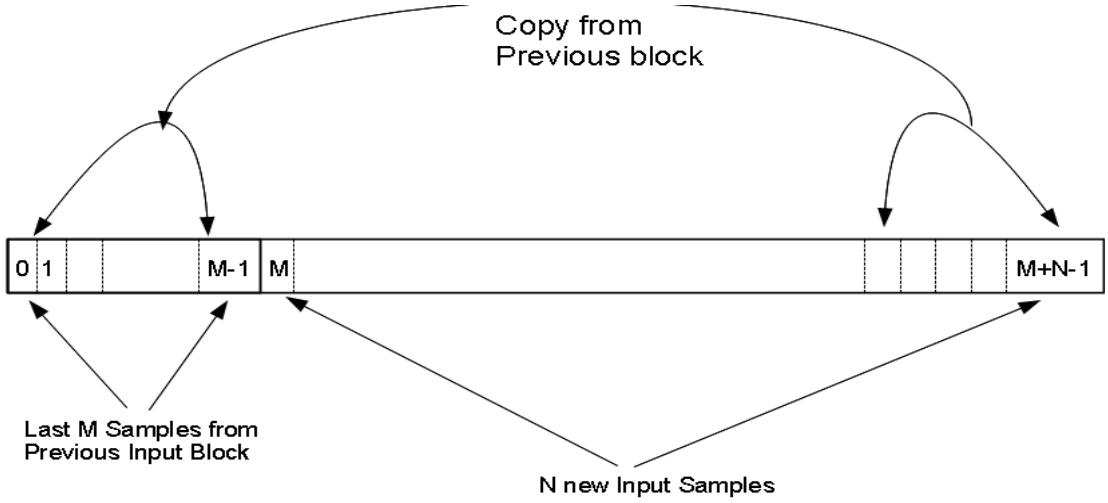
### 4.17.2 Implementation

Implementation is somewhat complicated due to keeping up with pointers and buffers as it traverses the input data and produces the output data stream.

The following is the code that generates the Sinc table with windowing. SINC_LENGTH is the total number of points in the table and SINC_PERIOD_PTS is the number of points per “Period” in the function.

```
for(i=0; i<SINC_LENGTH; i++)
{       //calc Blackman-Harris window points
    window = (0.35875
            - 0.48829*cos( (K_2PI*i)/(SINC_LENGTH-1) )
            + 0.14128*cos( (2.0*K_2PI*i)/(SINC_LENGTH-1) )
            - 0.01168*cos( (3.0*K_2PI*i)/(SINC_LENGTH-1) ) );
    //calculate sin(x)/x     sinc point * window
    fi = K_PI*(double)(i - SINC_LENGTH/2)/(double)SINC_PERIOD_PTS ;
    if(i != SINC_LENGTH/2)
            m_pSinc[i] = window * (TYPEREAL)sin( (double)fi )/(double)fi;
    else
            m_pSinc[i] = 1.0;
}
```

Input data is processed in blocks. First the new block of input sample data is copied into an input buffer after the last SINC_PERIODS(number of input sample periods in the Sinc function) samples from the previous block processed. This is the normal wrap around buffering required when doing convolution just like a FIR filter where you need to hang on to the last N samples to use in the next convolution of a data block.




```
//copy input samples into buffer starting at position SINC_PERIODS
j = SINC_PERIODS;
for(i=0; i<InLength; i++)
    m_pInputBuf[j++] = pInBuf[i];
```

The next step is to perform the convolution with the Sinc function stored in a table. Two time position variables are used(IntegerTime and m_FloatTime). One is a floating point variable that is incremented by a value that is the ratio of the input to output sample rate, and an integer time variable that points to the input sample position.

A while loop produces an output sample for every iteration and advances the floating output time variable by one output sample period. The input integer position is set to the truncated value of the floating point time variable and the loop continues until the integer input pointer reaches the last sample that it can calculate.

Finally the Float time is backed up to the beginning of the input buffer but retaining any fractional leftover. Then the last SINC_PERIODS of input data are copied back to the beginning of the input buffer for next time.

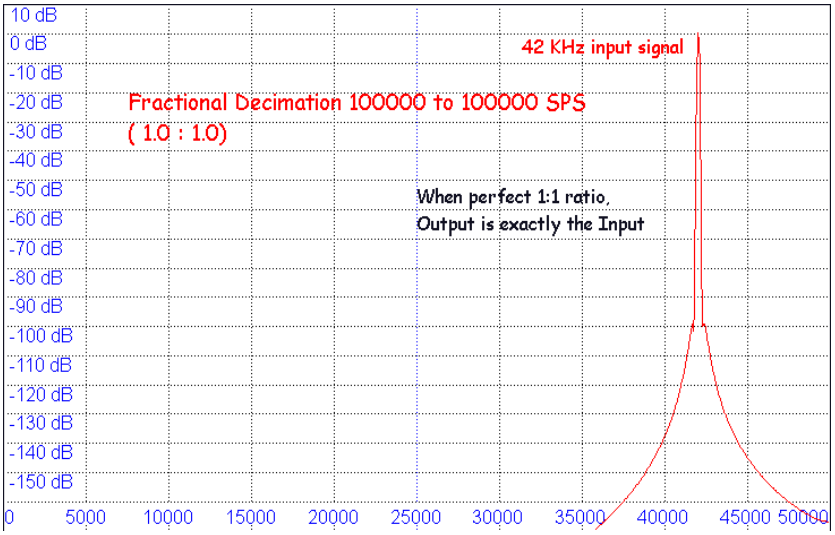
```
//now calculate output samples by looping until end of input buffer
// is reached.  The output position is incremented in fractional time
// of input sample time until all the possible input samples are
//processed.
double dt = Rate;       //output delta time as function of input sample time (input rate/output rate)
while(IntegerTime < InLength )
{   //convolve sinc function with input samples where sinc
    //function is centered at the output fractional time position
    acc = 0.0;
    for(i=1; i<=SINC_PERIODS; i++)
    {
            j = IntegerTime + i;    //temp integer time position for convolution loop
            int sindx =  (int)(( (double)j - m_FloatTime) * (double)SINC_PERIOD_PTS );
            acc += (m_pInputBuf[j].re * m_pSinc[sindx] );
    }
    pOutBuf[outsamples++] = acc;
    m_FloatTime += dt;
    IntegerTime = (int)m_FloatTime;
}

m_FloatTime -= (double)InLength;  //move floating time position back for next call
                                  //keeping leftover fraction
//need to copy last SINC_PERIODS input samples in buffer to beginning of buffer
// for FIR wrap around management. j points to last input sample processed
j = InLength;
for(i=0; i<SINC_PERIODS; i++)
    m_pInputBuf[i].re = m_pInputBuf[j++].re;
```

Several versions of this processing routine are implemented as overloaded functions to be able to use complex, real, short integer or stereo integer data.

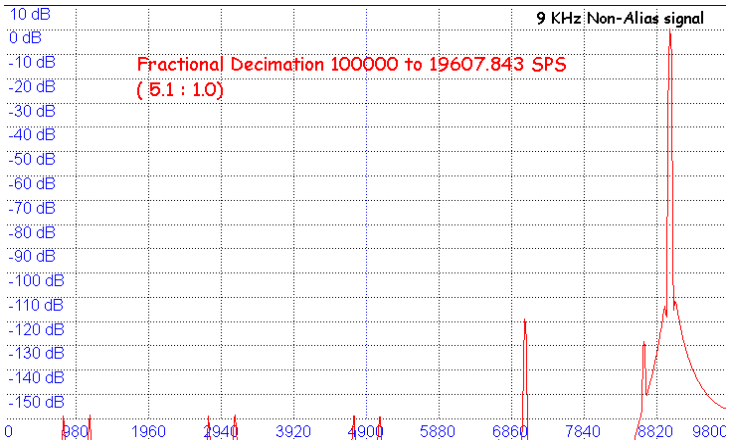
### 4.17.3 Fractional ReSampler Performance Testing.

The following is a test plot of a 42KHz 0dB full scale input signal with an exact one to one 100Ksps sample rate conversion.



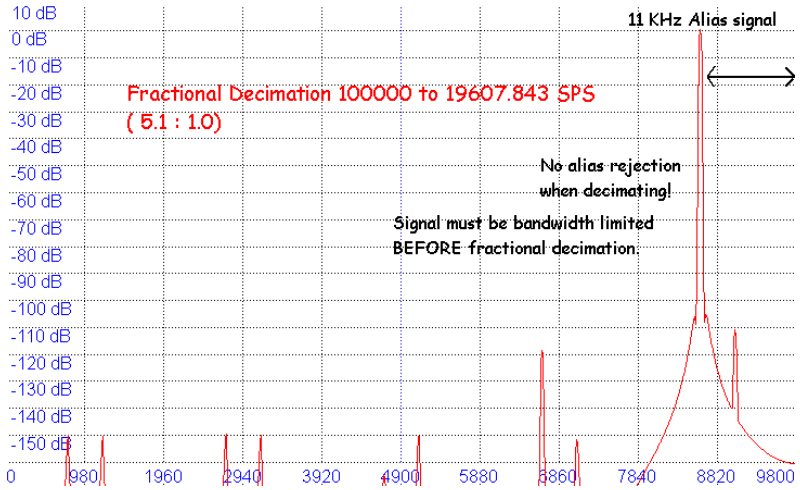

The output is exactly the input as expected. The shape of the signal spectrum is due to the windowed FFT used in the display.

Decimation or going from higher to a lower sample rate was examined. The resampler was set for a 5.1 to 1 ratio going from 100000 to 19608 SPS. A 9KHz input signal is shown below. The largest spurious signal is -120dB down. The Sinc function table had 28 periods with 10000 points between periods.

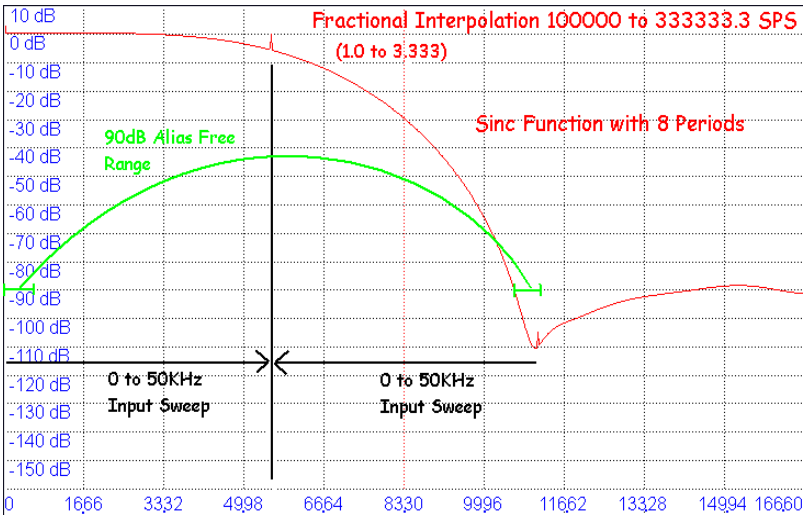


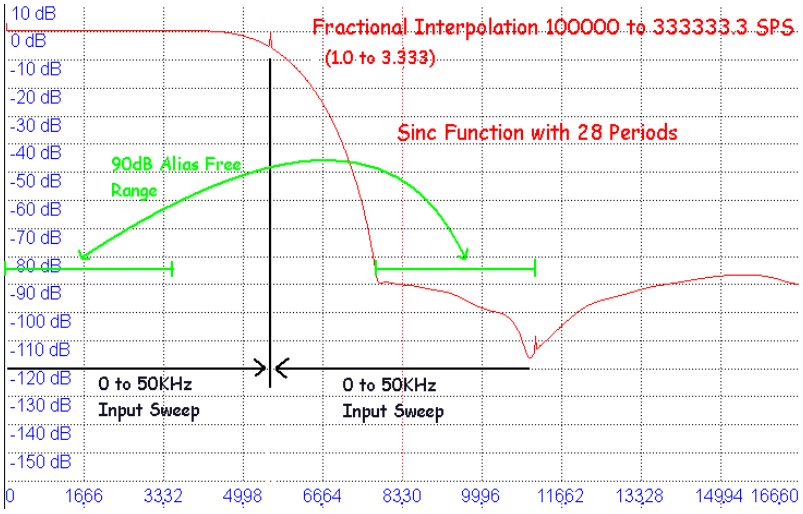
With the same setup, an 11 KHz input signal was applied which is above the Nyquist frequency for the output sample rate of 19608. Notice that the 11KHz signal has completely aliased back into the output signal with no attenuation. This is why it is important to bandwidth limit the input signal to one half the output sample rate when doing sample rate reduction. In CuteSDR this is done in the main bandpass filters and in some demodulation stages.




Fractional Interpolation or going from a low to higher sample rate was examined by sweeping an input signal from 0 to 50KHz with a 100000 SPS input sample rate and a 333333.3 SPS output rate(1 to 3.333 ratio). The Sinc function table had 8 periods with 10000 points between periods. As the input sweeps upward, an alias sweeps downward from 100KHz to 50KHz. In this case for 90dB of alias rejection, there is only about 5KHz of useable bandwidth.

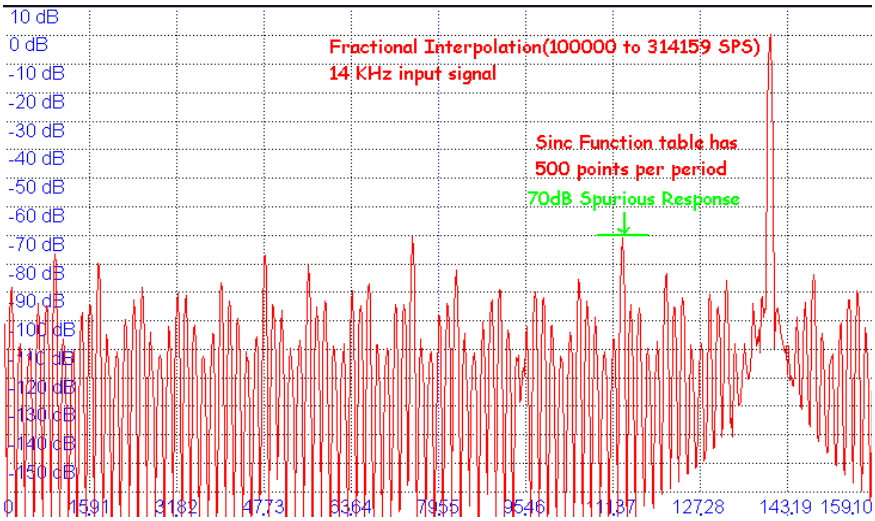


The number of Sinc function Periods is now increased to 28 and the same test run again. Note the 90dB alias free bandwidth has increased to over 30KHz. By adjusting the number of Sinc function periods, one can optimize the alias free bandwidth for the range of sample rates desired.

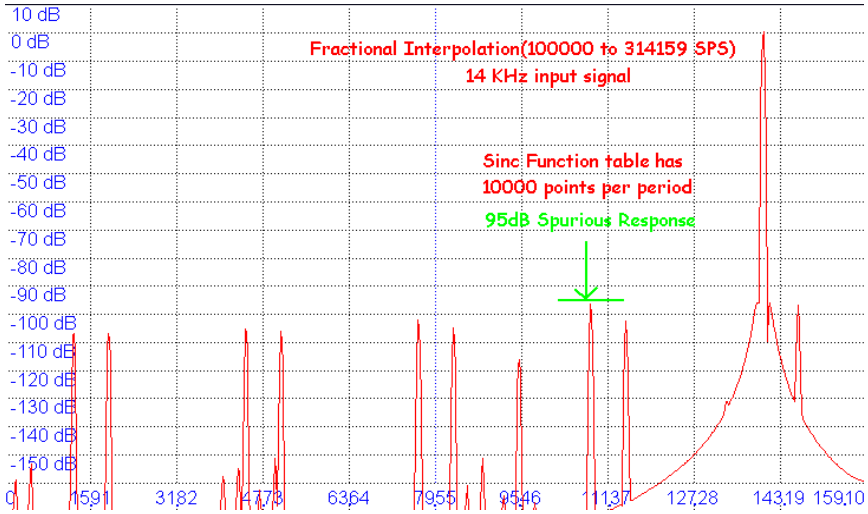


The resampler spurious response was examined by interpolating by PI(3.14159) going from 100KSps to 314.159 KSps. The Sinc function table had 28 periods with 500 points between periods. A 14KHz input signal was applied. Note the high level of spurious signals within the spectrum.




The same setup but with 10000 points between periods in the Sinc table gives much better spurious response.




If the resampling was performed before the high dynamic range demodulation stages, it can be seen that there could be visible and audible artifacts generated by the resampling process. By placing it after the demodulators and AGC and performing the resampling on the audio signal, one can tolerate spurious signals that are 90 dB down since the ear has less dynamic range and the artifacts are not audible. A side benefit as will be shown later in the sound card output section, is that this resampler can also be used to dynamically track the clock differences between the SDR input sample rate and the PC soundcard output sample rate.

## *4.18. General Purpose FIR Filter (CFir Class)*

The CFir class is used to create and implement small FIR filters for general purpose filtering. It can take pre-computed coefficients or can design simple filters using specifications of pass and stop bands.

### 4.18.1 Implementation

There are several ways to implement a FIR filter and CuteSDR uses one that has a coefficient table that is twice the length of the number of coefficients. This is done to eliminate a wrap around test in the inner loop of the algorithm that is performing the "MAC" operations. A ring buffer, m_ZBuf is used to hold the input samples and a state variable m_State keeps track of current position. Local automatic variables, Zptr and Hptr are used as pointers to the data and coefficients for efficiency. Note that in the inner for loop there is no need to test for the end of the coefficient buffer since it is twice as long with duplicate coefficients.

```
for(int i=0; i<InLength; i++)
{
	m_rZBuf[m_State] = InBuf[i];
	Hptr = &m_Coef[m_NumTaps - m_State];
	Zptr = m_rZBuf;
	acc = (*Hptr++ * *Zptr++);	//do the 1st MAC
	for(int j=1; j<m_NumTaps; j++)
		acc += (*Hptr++ * *Zptr++);	//do the remaining MACs
	if(--m_State < 0)
		m_State += m_NumTaps;
	OutBuf[i] = acc;
}
```

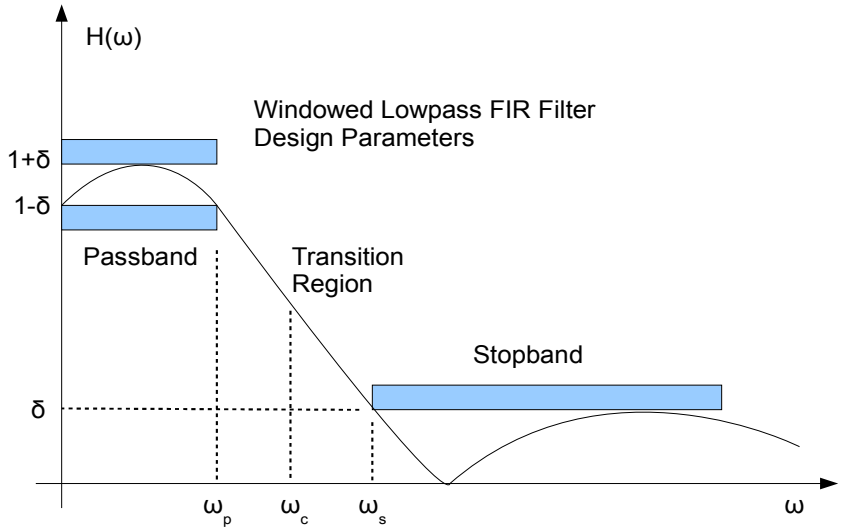
For pre calculated coefficients, the following initialization function is used that just copies the users coefficients into the classes coefficient buffer twice and initializes the buffers.

```
////////////////////////////////////////////////////////////////////////////
//  Initializes a pre-designed FIR filter with fixed coefficients
//	Initialize FIR variables and clear out buffers.
////////////////////////////////////////////////////////////////////////////
void CFir::InitConstFir( int NumTaps, const double* pCoef)
{
    for(int i=0; i<m_NumTaps; i++)
    {
		m_Coef[i] = pCoef[i];
		m_Coef[m_NumTaps+i] = pCoef[i]; //create duplicate for calculation efficiency
    }
    for(int i=0; i<m_NumTaps; i++)
    {		//zero input buffers
		m_rZBuf[i] = 0.0;
		m_cZBuf[i].re = 0.0;
		m_cZBuf[i].im = 0.0;
    }
    m_State = 0;	//zero filter state variable
}
```

### 4.18.2 Kaiser-Bessel Filter Design Method

For designing filter coefficients on the fly, this class uses the Kaiser windowing method. This method was chosen since it provides not only the windowing function, but also returns an estimate for how many coefficients (or FIR taps) are required. Normally filter design programs require a lot of trial and error trying different windows and number of coefficients to meet the required filter specification. Typically one trades off transition band sharpness, stopband gain, and passband ripple. The Kaiser window does not produce the best window for all cases but does work well for a lot of general filtering needs.

The filter specifications for the Kaiser method are transition region width ($\omega_s - \omega_p$) and stopband gain($\delta$). Note that the normal 6dB lowpass cutoff frequency is $\omega_c = (\omega_s + \omega_p)/2$.




Note that as the stopband gain gets smaller(more attenuation) that the passband ripple also gets smaller and for most filters will be insignificant.

The Kaiser window filter design method calculates two parameters, β the shape factor and M the number of coefficients minus 1 from the filter specifications of ($\omega_s - \omega_p$) and stopband gain($\delta$).

Let A == stopband gain in dB == $-20\log_{10}(\delta)$.
and $\Delta\omega == (\omega_s - \omega_p)$

The lowpass cutoff frequency is $\omega_c = (\omega_s + \omega_p)/2$

β is determined empirically using the following Kaiser-Bessel equations:

If A>50 then

$$\beta = 0.1102(A - 8.71)$$

If 20.96 ≤ A ≤ 50 then

$$\beta = 0.5842(A - 20.96)^{0.4} + 0.7886(A - 20.96)$$

If A < 20.96 then

$\beta = 0.0$ (rectangular window, all one's)

M is the number of coefficients required -1 and is equal to:

$$M = \frac{A-8}{2.285\,\Delta\omega}$$

The ideal lowpass filter coefficients are calculated using the Sinc function multiplied by the Kaiser window function:

$$h[n] = \frac{\sin(\omega_c(n-M/2))}{\pi(n-M/2)} \cdot \frac{I_0\left(\beta\sqrt{1-\left[\frac{n-(M/2)}{M/2}\right]^2}\right)}{I_0(\beta)} \qquad 0 \le n \le M$$

ideal Lowpass Filter Kaiser Window

where $I_0$ is a modified Bessel function of zero order of the first kind or a $I_0$-Sinh function. Bessel functions have no closed form way of calculating so either approximations or using iterative methods must be used. The following formula can be used by incrementing k from 1 till the difference between the present and last value is less than some small error limit.

$$I_0(x) = 1 + \sum_{k=1}^{\infty}\left[\frac{(x/2)^k}{k!}\right]^2$$

The following code calculates $I_0$ using an iterative loop until the error is less than a specified value.

```
TYPEREAL CFir::Izero(TYPEREAL x)
{
TYPEREAL x2 = x/2.0;
TYPEREAL sum = 1.0;
TYPEREAL ds = 1.0;
TYPEREAL di = 1.0;
TYPEREAL errorlimit = 1e-9;
TYPEREAL tmp;
    do
    {
            tmp = x2/di;
            tmp *= tmp;
            ds *= tmp;
            sum += ds;
            di += 1.0;
    }while(ds >= errorlimit*sum);
    return(sum);
}
```

### 4.18.3 Low Pass Implementation

The following is the code segment that creates the coefficients for a low pass filter.

```
int CFir::InitLPFilter(TYPEREAL Scale, TYPEREAL Astop,
                TYPEREAL Fpass, TYPEREAL Fstop, TYPEREAL Fsamprate)
{
int n;
TYPEREAL Beta;
m_Mutex.lock();
m_SampleRate = Fsamprate;
//create normalized frequency parameters
TYPEREAL normFpass = Fpass/Fsamprate;
TYPEREAL normFstop = Fstop/Fsamprate;
TYPEREAL normFcut = (normFstop + normFpass)/2.0;        //low pass filter 6dB cutoff
    //calculate Kaiser-Bessel window shape factor, Beta, from stopband attenuation
    if(Astop < 20.96)
                Beta = 0;
    else if(Astop >= 50.0)
                Beta = .1102 * (Astop - 8.71);
    else
                Beta = .5842 * pow( (Astop-20.96), 0.4) + .07886 * (Astop - 20.96);
//Now Estimate number of filter taps required based on filter specs
    m_NumTaps = (Astop - 8.0) / (2.285*K_2PI*(normFstop - normFpass) ) + 1;
//clamp range of filter taps
    if(m_NumTaps>MAX_NUMCOEF )
                m_NumTaps = MAX_NUMCOEF;
    if(m_NumTaps < 3)
                m_NumTaps = 3;
    TYPEREAL fCenter = .5*(TYPEREAL)(m_NumTaps-1);
    TYPEREAL izb = Izero(Beta);             //precalculate denominator since is same for all points
    for( n=0; n < m_NumTaps; n++)
    {
                TYPEREAL x = (TYPEREAL)n - fCenter;
                TYPEREAL c;
                // create ideal Sinc() LP filter with normFcut
                if( (TYPEREAL)n == fCenter )    //deal with odd size filter singularity where sin(0)/0==1
                            c = 2.0 * normFcut;
                else
                            c = (TYPEREAL)sin(K_2PI*x*normFcut)/(K_PI*x);
                //calculate Kaiser window and multiply to get coefficient
                x = ((TYPEREAL)n - ((TYPEREAL)m_NumTaps-1.0)/2.0 ) / (((TYPEREAL)m_NumTaps-1.0)/2.0);
                m_Coef[n] = Scale * c * Izero( Beta * sqrt(1 - (x*x) ) )  / izb;
    }
}
```

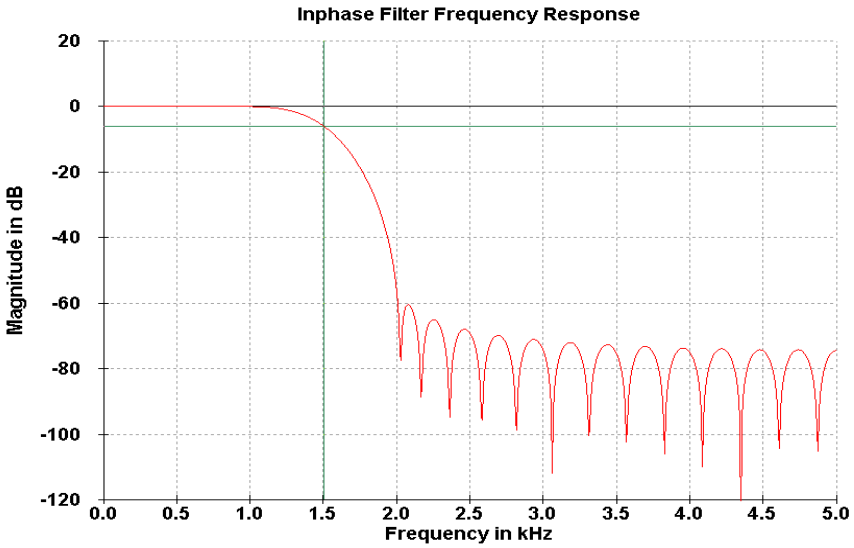
### 4.18.4 Low Pass Filter Verification

The following filter was designed to test the implementation of a lowpass filter using the CFir class.
Sample rate = 10KHz
Fpassband = 1KHz
Fstopband = 2KHz
Stopband Attenuation = 60dB.

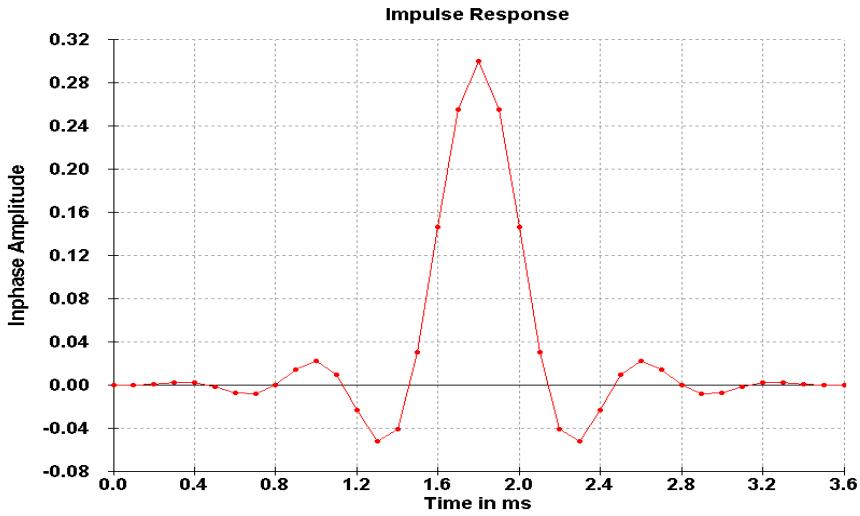
The filter is designed with this function call:
```
m_Fir.InitLPFilter(1.0, 60.0, 1000.0, 2000.0, 10000.0);//initialize LP FIR filter
```

The number of coefficients required was 37.

The following are plots of the coefficients and the magnitude frequency response. The green cursor position shows the 6dB lowpass cutoff frequency of 1.5KHz. [ (1+2) /2 ]




The impulse response is just the plot of the 37 FIR coefficients.

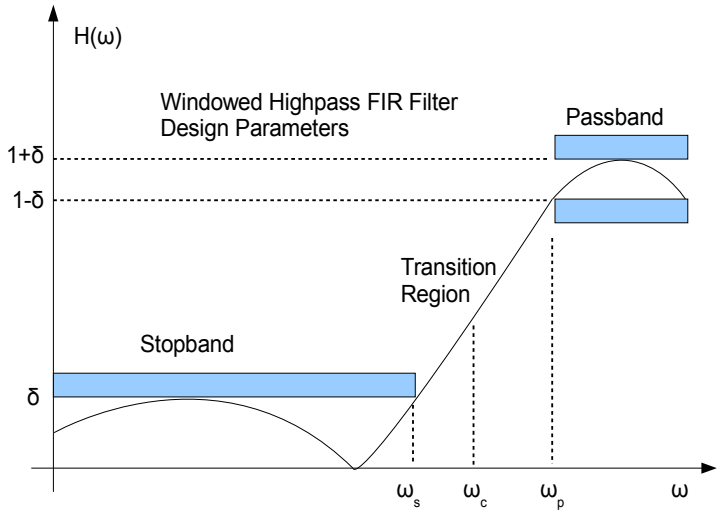


### 4.18.5 High Pass Implementation

A high pass filter can also be designed using the same basic technique using the following somewhat busy formula.

$$h[n] = \left\{ \frac{\sin(\pi(n-M/2))}{\pi(n-M/2)} - \frac{\sin(\omega_c(n-M/2))}{\pi(n-M/2)} \right\} \cdot \frac{I_0\left( \beta\sqrt{1-[\frac{n-(M/2)}{M/2}]^2} \right)}{I_0(\beta)}$$

The filter parameters are similar to the Low pass filter and the Kaiser-Bessel parameters calculated the same way.




A separate function can be called in the CFir class to create a High Pass filter.
This is the code segment that calculates the coefficients.

```
TYPEREAL fCenter = .5*(TYPEREAL)(m_NumTaps-1);
for( n=0; n < m_NumTaps; n++)
{
	TYPEREAL x = (TYPEREAL)n - (TYPEREAL)(m_NumTaps-1)/2.0;
	TYPEREAL c;
	// create ideal Sinc() HP filter with normFcut
	if( (TYPEREAL)n == fCenter )    //deal with odd size filter singularity where sin(0)/0==1
		c = 1.0 - 2.0 * normFcut;
	else
		c = (TYPEREAL) (sin(K_PI*x)/(K_PI*x) - sin(K_2PI*x*normFcut)/(K_PI*x) );
	//calculate Kaiser window and multiply to get coefficient
	x = ((TYPEREAL)n - ((TYPEREAL)m_NumTaps-1.0)/2.0 ) / (((TYPEREAL)m_NumTaps-1.0)/2.0);
	m_Coef[n] = Scale * c * Izero( Beta * sqrt(1 - (x*x) ) )  / izb;
}
```

A subtle difference between the low pass and high pass filters is that the high pass filters are forced to an odd number of taps. This gives better response on the high pass filter as explained in “Discrete-Time Signal Processing” by Oppenheim and Schafer pg 479-481. An even number of coefficients creates a zero in H(z) at ω0 = π so the high pass response dips to zero right at the Nyquist frequency.

### 4.18.6 High Pass Verification

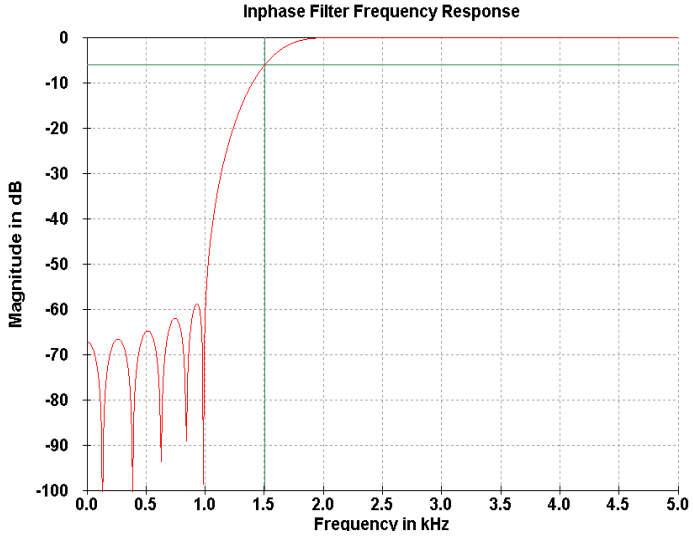
The following filter was designed to test the implementation of a highpass filter using the CFir class.
Sample rate = 10KHz
Fpassband = 2KHz
Fstopband = 1KHz
Stopband Attenuation = 60dB.

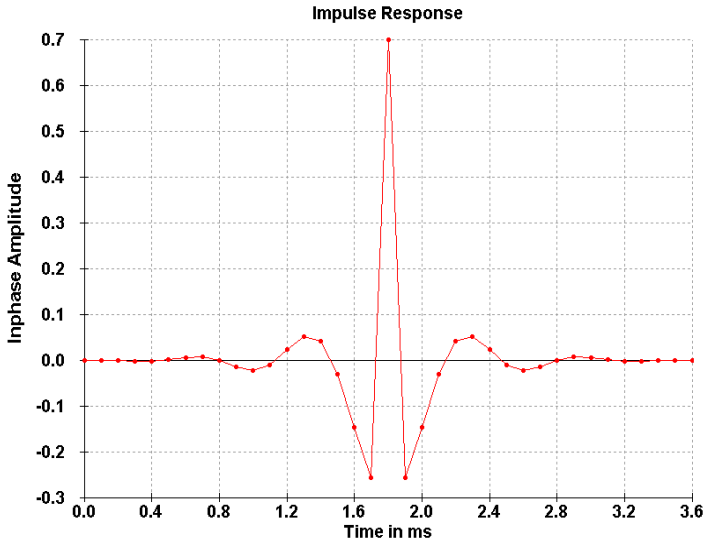
The filter is designed with this function call:

```
m_Fir.InitHPFilter(1.0, 60.0, 2000.0, 1000.0, 10000.0);//initialize HP FIR filter
```

The number of coefficients required was 37.

The following are plots of the coefficients and the magnitude frequency response. The green cursor position shows the 6dB highpass cutoff frequency of 1.5KHz. [ (1+2) /2 ]



### 4.18.7 Hilbert Filter Pair Generation

The CFir also implements a function to convert a low pass filter into a complex band pass filter with 90 degree phase shift between the I and Q filters. This is useful for selecting upper or lower sideband from an I/Q data stream. This is used in the Synchronous AM detector to split the LSB and USB into the left and right audio channels.

The procedure is to design a low pass filter with the cutoff frequency around ½ the required bandwidth of the desired final audio bandwidth. Once the filter coefficients are found, the following equations are used to both shift the filters in frequency and generate a phase shift between them of 90 degrees.

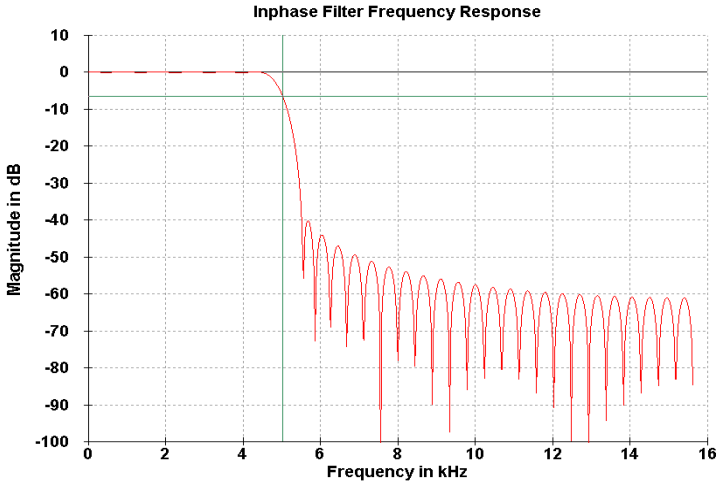
$$hi_{BP}(n) = 2h_{LP}(n)\cos\left(2\pi f_0\left[n - \frac{(N-1)}{2}\right]T\right) \qquad hq_{BP}(n) = 2h_{LP}(n)\sin\left(2\pi f_0\left[n - \frac{(N-1)}{2}\right]T\right)$$

where:
$h_{LP}$[] are the original Lowpass filter coefficients,
$hi_{BP}$[] are the I channel bandpass filter coefficients,
$hq_{BP}$[] are the Q channel bandpass filter coefficients,
N is the number of filter coefficients,
$f_0$ is the filter frequency shift in Hz,
T is the sample period,
and n is the coefficient index from 0 to N-1.

A C code snippet from the CFir class to perform the transform is:

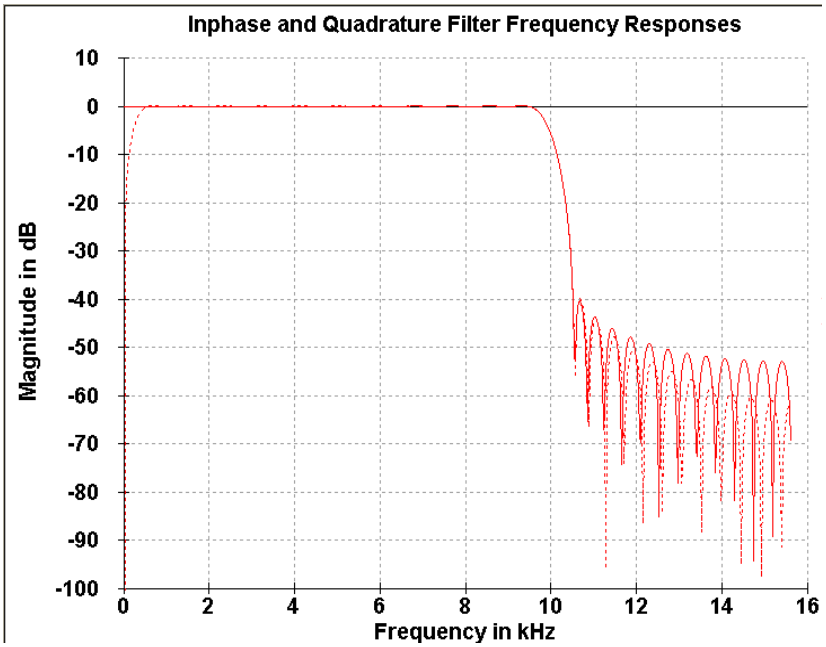
```
for(n=0; n<m_NumTaps; n++)
{
    // apply complex frequency shift transform to low pass filter coefficients
    m_ICoef[n] = 2.0 * m_Coef[n] * cos( (K_2PI*FreqOffset/m_SampleRate)*(n-( (m_NumTaps-1)/2 ) ) );
    m_QCoef[n] = 2.0 * m_Coef[n] * sin( (K_2PI*FreqOffset/m_SampleRate)*(n-( (m_NumTaps-1)/2 ) ) );
}
```

The following example is from the CFir class which can be used to create and implement the I and Q Hilbert Band Pass filter pairs.
First a Low Pass filter is created with the following parameters:
Pass band cutoff = 4500Hz, Stop band frequency = 5500Hz, Stop band attenuation 40dB,
and Sample rate = 31250Hz.
The designed filter contained 70 taps and its frequency response is plotted below:



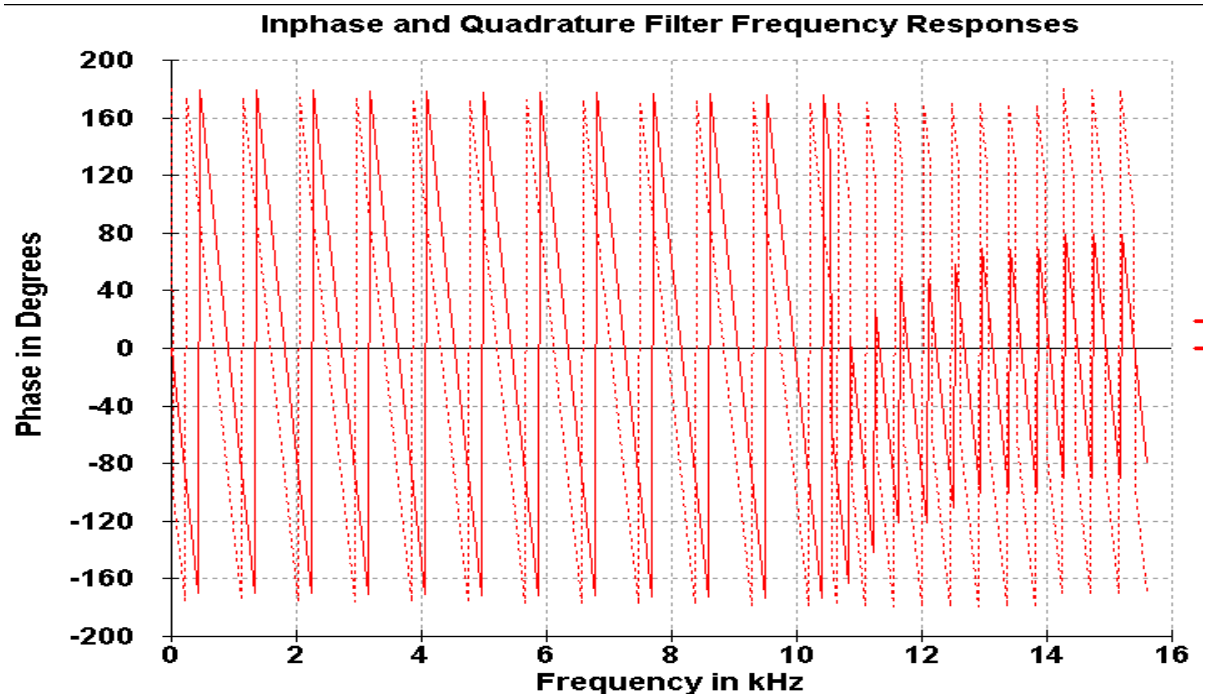

  

Next the Hilbert Band pass pair was created using an offset of 5000Hz. This shifted the lowpass into a 10000Hz band pass with the following magnitude response:




Note the magnitudes are equal in the pass band.

The phase plot of both filters is below showing a 90 degree difference between the I and Q filters within the band pass of the filters:



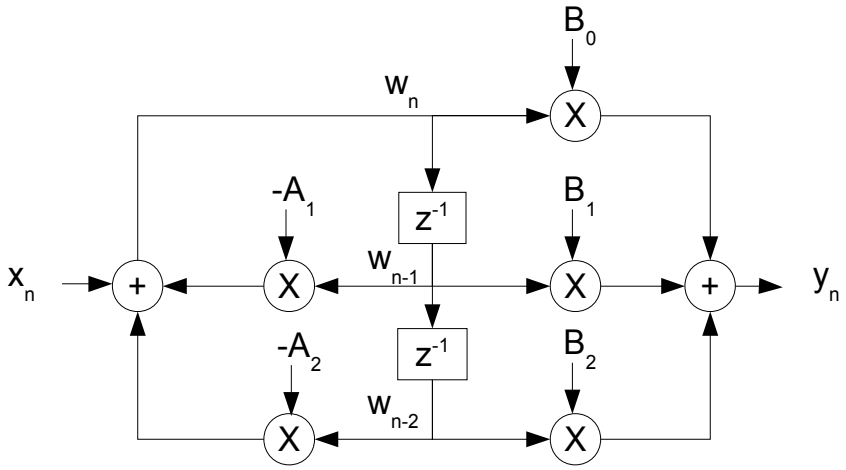
## *4.19. IIR Filters (Clir Class)*

### 4.19.1 Design

The Clir class implements a second order IIR filter stage. The transfer function for this filter is called Direct 2 form :

$$H(z)=\frac{B_0+B_1 z^{-1}+B_2 z^{-2}}{1+A_1 z^{-1}+A_2 z^{-2}}$$

The implementation of this function in block form is




2nd order IIR Direct 2 Form $H(z)=\frac{B_0+B_1 z^{-1}+B_2 z^{-2}}{1+A_1 z^{-1}+A_2 z^{-2}}$

This is easily implemented in C code as:

```
for(int i=0; i<InLength; i++)
{
	TYPEREAL w0 = InBuf[i] - m_A1*m_w1a - m_A2*m_w2a;
	OutBuf[i] =m_B0*w0 + m_B1*m_w1a + m_B2*m_w2a;
	m_w2a = m_w1a;
	m_w1a = w0;
}
```

The hard part is coming up with the 5 constants to make a stable filter. Unlike FIR filters, IIR filters can be unstable since they have feedback from the output to the input. It is beyond the scope of this paper to get into the details of IIR filter design as there are many books and papers on the subject. In CuteSDR, a cookbook approach is used from a paper *"Cookbook formulae for audio EQ biquad filter coefficients" by Robert Bristow-Johnson*. He gives a nice set of equations for the IIR constants based on essentially 2 input parameters Q and Frequency. Frequency depends on the type of filter so can be lowpass corner frequency or center frequency or as the paper states "wherever its happenin man"

The most common design approach is to design an analog filter then convert it to a digital filter in the z domain. The method used in this paper is using the bi-linear transform with compensation for frequency warping. Basically one substitutes the following equation for 's' in the analog filter transfer function.

$$s=\frac{1}{\tan(W_0/2)}\cdot\frac{1-z^{-1}}{1+z^{-1}}$$

Substituting this into the H(s) filter equations can be seen to be quite messy so the cookbook approach is a welcome outcome for those not wishing to dive into a lot of algebra and trig identities.

Four filter types are supported by the Clir class. A lowpass, highpass, bandpass, and band reject filter. Each is initialized by a separate function call with the parameters of F0Freq, FilterQ, and SampleRate.

### 4.19.2 Low Pass Implementation

The low pass filter uses the following equations for determining the IIR constants.

The analog filter prototype is $H(s)=\frac{1}{s^2+s/Q+1}$

and the IIR filter coefficients are:

let $\omega_0=2\pi\, F0Freq$

$$alpha=\frac{\sin(\omega_0)}{2\cdot FilterQ}$$

$A=\frac{1}{1+alpha}$ scale all constants by this value for direct form 2

$$B_0=\frac{A(1-\cos(\omega_0))}{2}$$

$$B_1=A(1-\cos(\omega_0))$$

$$B_2=\frac{A(1-\cos(\omega_0))}{2}$$

$$A_1=-2A\cos(\omega_0)$$

$$A_2=A(1-alpha)$$

The C code to calculate the LP filter coefficients in Clir is:

```
void Clir::InitLP( TYPEREAL F0Freq, TYPEREAL FilterQ, TYPEREAL SampleRate)
{
	TYPEREAL w0 = K_2PI * F0Freq/SampleRate;	//normalized corner frequency
	TYPEREAL alpha = sin(w0)/(2.0*FilterQ);
	TYPEREAL A = 1.0/(1.0 + alpha); //scale everything by 1/A0 for direct form 2
	m_B0 = A*( (1.0 - cos(w0))/2.0);
	m_B1 = A*( 1.0 - cos(w0));
	m_B2 = A*( (1.0 - cos(w0))/2.0);
	m_A1 = A*( -2.0*cos(w0));
	m_A2 = A*( 1.0 – alpha);
}
```

### 4.19.3 High Pass Implementation

The high pass filter uses the following equations for determining the IIR constants.

The analog filter prototype is $H(s)=\frac{s^2}{s^2+s/Q+1}$

and the IIR filter coefficients are:

let $\omega_0=2\pi\, F0Freq$

$$alpha=\frac{\sin(\omega_0)}{2\cdot FilterQ}$$

$A=\frac{1}{1+alpha}$ scale all constants by this value for direct form 2

$$B_0=\frac{A(1+\cos(\omega_0))}{2}$$

$$B_1=-A(1+\cos(\omega_0))$$

$$B_2=\frac{A(1+\cos(\omega_0))}{2}$$

$$A_1=-2A\cos(\omega_0)$$

$$A_2=A(1-alpha)$$

The C code to calculate the HP filter coefficients in CIir is:

```
void CIir::InitHP( TYPEREAL F0Freq, TYPEREAL FilterQ, TYPEREAL SampleRate)
{
	TYPEREAL w0 = K_2PI * F0Freq/SampleRate;	//normalized corner frequency
	TYPEREAL alpha = sin(w0)/(2.0*FilterQ);
	TYPEREAL A = 1.0/(1.0 + alpha); //scale everything by 1/A0 for direct form 2
	m_B0 = A*( (1.0 + cos(w0))/2.0);
	m_B1 = -A*( 1.0 + cos(w0));
	m_B2 = A*( (1.0 + cos(w0))/2.0);
	m_A1 = A*( -2.0*cos(w0));
	m_A2 = A*( 1.0 – alpha);
}
```

### 4.19.4 Band Pass Implementation

The band pass filter uses the following equations for determining the IIR constants.

The analog filter prototype is $H(s)=\frac{s/Q}{s^2+s/Q+1}$

and the IIR filter coefficients are:

let $\omega_0=2\pi\, F0Freq$

$$alpha=\frac{\sin(\omega_0)}{2\cdot FilterQ}$$

$A=\frac{1}{1+alpha}$ scale all constants by this value for direct form 2

$$B_0=A\cdot alpha$$

$$B_1=0$$

$$B_2=-A\cdot alpha$$

$$A_1=-2A\cos(\omega_0)$$

$$A_2=A(1-alpha)$$

The C code to calculate the BP filter coefficients in CIir is:

```
void CIir::InitBP( TYPEREAL F0Freq, TYPEREAL FilterQ, TYPEREAL SampleRate)
{
	TYPEREAL w0 = K_2PI * F0Freq/SampleRate;	//normalized corner frequency
	TYPEREAL alpha = sin(w0)/(2.0*FilterQ);
	TYPEREAL A = 1.0/(1.0 + alpha); //scale everything by 1/A0 for direct form 2
	m_B0 = A * alpha;
	m_B1 = 0.0;
	m_B2 = A * -alpha;
	m_A1 = A*( -2.0*cos(w0));
	m_A2 = A*( 1.0 - alpha);
}
```

### 4.19.5 Band Reject (Notch) Implementation

The band reject(or Notch) filter uses the following equations for determining the IIR constants.

The analog filter prototype is $H(s)=\frac{s^2+1}{s^2+s/Q+1}$

and the IIR filter coefficients are:

let $\omega_0=2\pi\, F0Freq$

$$alpha=\frac{\sin(\omega_0)}{2\cdot FilterQ}$$

$A=\frac{1}{1+alpha}$ scale all constants by this value for direct form 2

$$B_0=A$$

$$B_1=-2\mathrm{A}\cos(\omega_0)$$

$$B_2=A$$

$$A_1=-2\mathrm{A}\cos(\omega_0)$$

$$A_2=A(1-alpha)$$

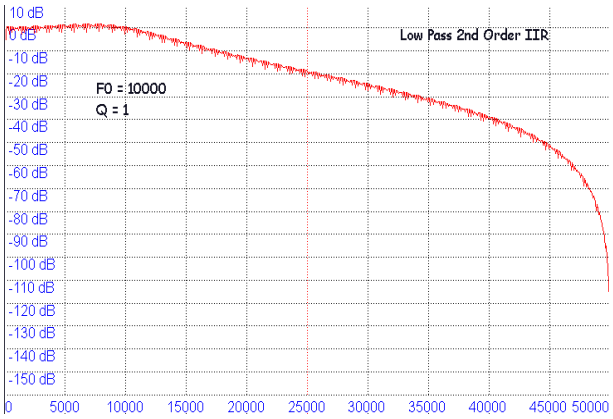
The C code to calculate the BR filter coefficients in CIir is:

```
void CIir::InitBR( TYPEREAL F0Freq, TYPEREAL FilterQ, TYPEREAL SampleRate)
{
	TYPEREAL w0 = K_2PI * F0Freq/SampleRate;	//normalized corner frequency
	TYPEREAL alpha = sin(w0)/(2.0*FilterQ);
	TYPEREAL A = 1.0/(1.0 + alpha); //scale everything by 1/A0 for direct form 2
	m_B0 = A*1.0;
	m_B1 = A*( -2.0*cos(w0));
	m_B2 = A*1.0;
	m_A1 = A*( -2.0*cos(w0));
	m_A2 = A*( 1.0 - alpha);
}
```

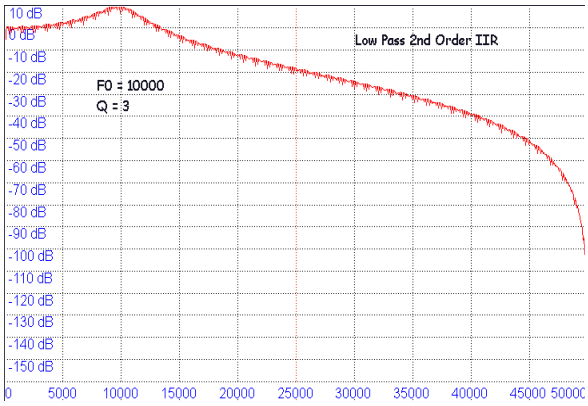
### 4.19.6 Filter Verification

The IIR filters were tested using the CuteSDR test bench sweep generator and plotting the resulting magnitude response. All tests used a sample rate of 100000 and an sweep input magnitude of 0dB. Plots with different Q values are plotted to show the affect of Q on the filter shapes.

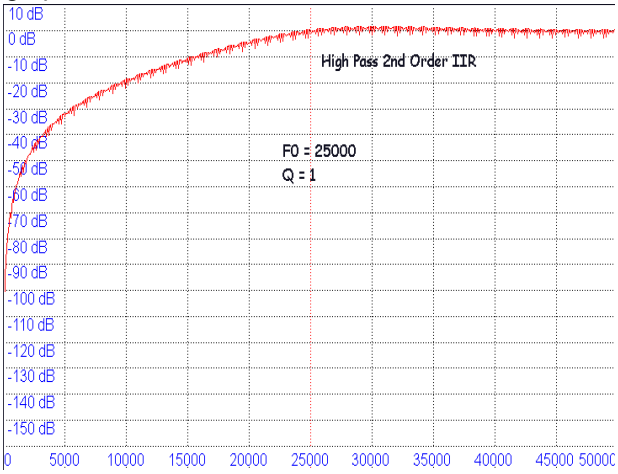
A low pass filter was created with an F0 of 10000 and a Q of 1.



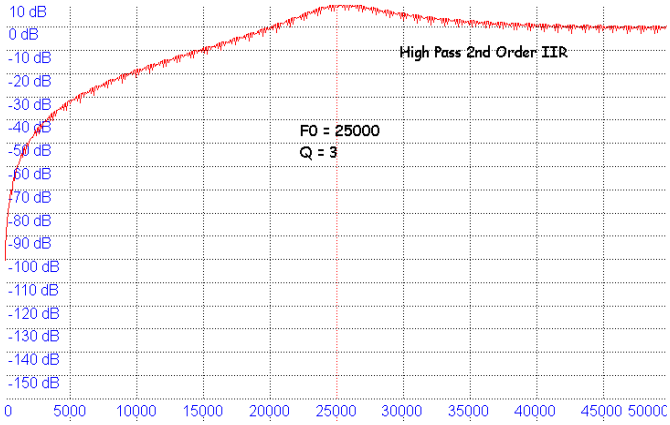

A low pass filter was created with an F0 of 10000 and a Q of 3. Note the peaking at F0 as Q is increased.



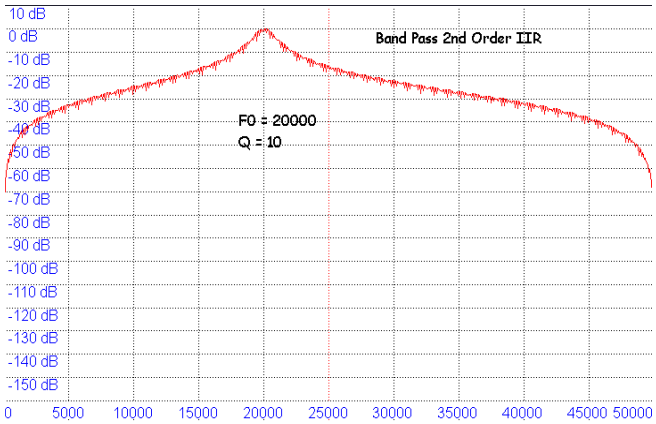

A high pass filter was created with an F0 of 25000 and a Q of 1.



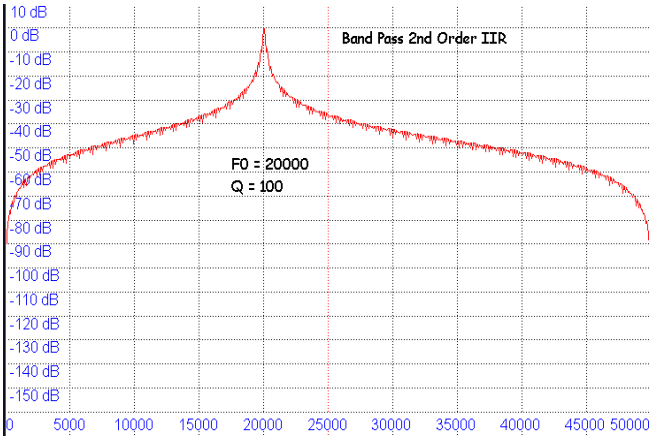
A high pass filter was created with an F0 of 25000 and a Q of 3. Again there is peaking as the Q increases.



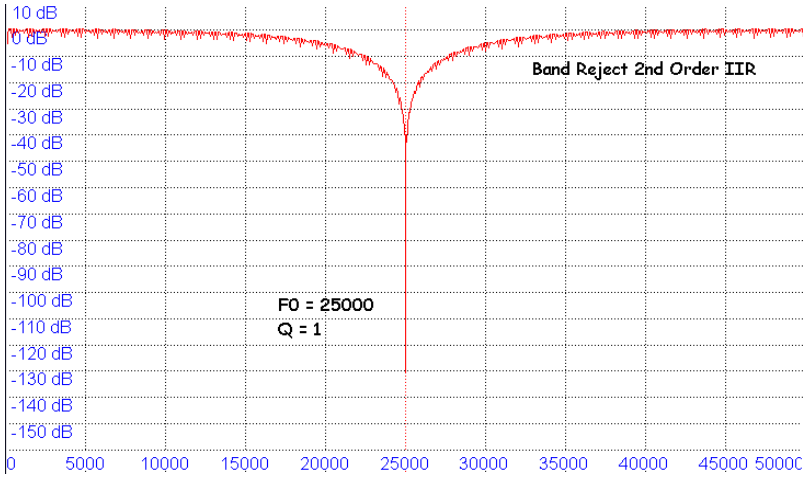

A band pass filter was created with an F0 of 20000 and a Q of 10.



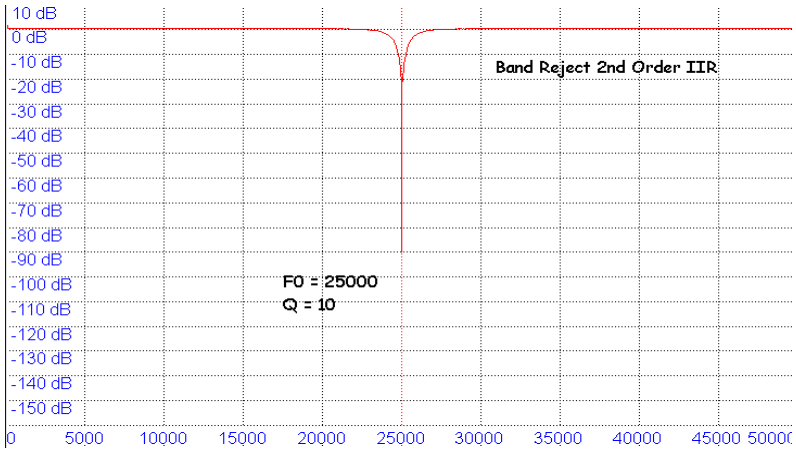

A band pass filter was created with an F0 of 20000 and a Q of 100.



A band reject of notch filter was created with an F0 of 25000 and a Q of 1. The depth of the notch is essentially infinite even though the plot does not show it due to FFT artifacts. With a double precision floating point implementation, very narrow deep accurate notches can be created by making Q large.




A band reject of notch filter was created with an F0 of 25000 and a Q of 10.



## *4.20. Sound Card Output (CSoundOut Class)*

### 4.20.1 Requirements

The CuteSDR design goal was to use the stock Qt sound card interface and not rely on 3rd party libraries. The QAudioOutput class is the only native resource that can be used for streaming audio out the sound card. It uses the Qt Signal-Slot paradigm to inform the data source process that new data can be sent to the soundcard. The problem found with this method is that the Slot function that receives the Signal appears to only work in the main GUI process thread. All sorts of things were tried to move it to a worker thread with no success. The problem with using the GUI thread for processing streaming data is that if it gets busy, the sound card data stream gets starved and the audio breaks up. Even the few program examples provided by Qt suffer from this problem and break up if you click on the window to move it etc.

A kludge to get around this problem was to not use the Signal-Slot scheme and just poll the QAudioOutput class's buffer to see when it is OK to give it more data. This is not efficient but does separate the GUI thread from the audio processing thread.

### 4.20.2 Implementation

The CuteSDR CSoundOut class inherits from the QThread class so it can implement the polling worker thread to manage the data flow into the soundcard. The sound card is run at a fixed rate of 48KHz but can be changed with a define to 44.1KHz if desired.

Several Qt classes are involved with setup of the soundcard.

QAudioDeviceInfo is used to select the desired sound card and is selected by the GUI code.
QAudioFormat is a structure that one fills in with the desired rate, sample size, nimber of channels etc.
QAudioOutput is the sound card output class used by Qt to manage the soundcard.

Some setup code is shown below:

```
//Setup fixed format for sound ouput
m_OutAudioFormat.setCodec("audio/pcm");
m_OutAudioFormat.setFrequency(SOUNDCARD_RATE);
m_OutAudioFormat.setSampleSize(16);
m_OutAudioFormat.setSampleType(QAudioFormat::SignedInt);
m_OutAudioFormat.setByteOrder(QAudioFormat::LittleEndian);
if(m_StereoOut)
      m_OutAudioFormat.setChannels(2);
else
      m_OutAudioFormat.setChannels(1);
m_pAudioOutput = new QAudioOutput(m_OutDeviceInfo, m_OutAudioFormat, this);
```

To begin running, the start() function is called. It returns a pointer to the QAudioOutput class's internal QIODevice object that one can use to perform low level data writes to. Then the start(..) for the CSoundOut worker thread that will poll the QAudioOutput object to know when it can write data to it.

```
m_pOutput = m_pAudioOutput->start(); //start Qt AudioOutput
start(QThread::HighestPriority);       //start worker thread and set its priority
```

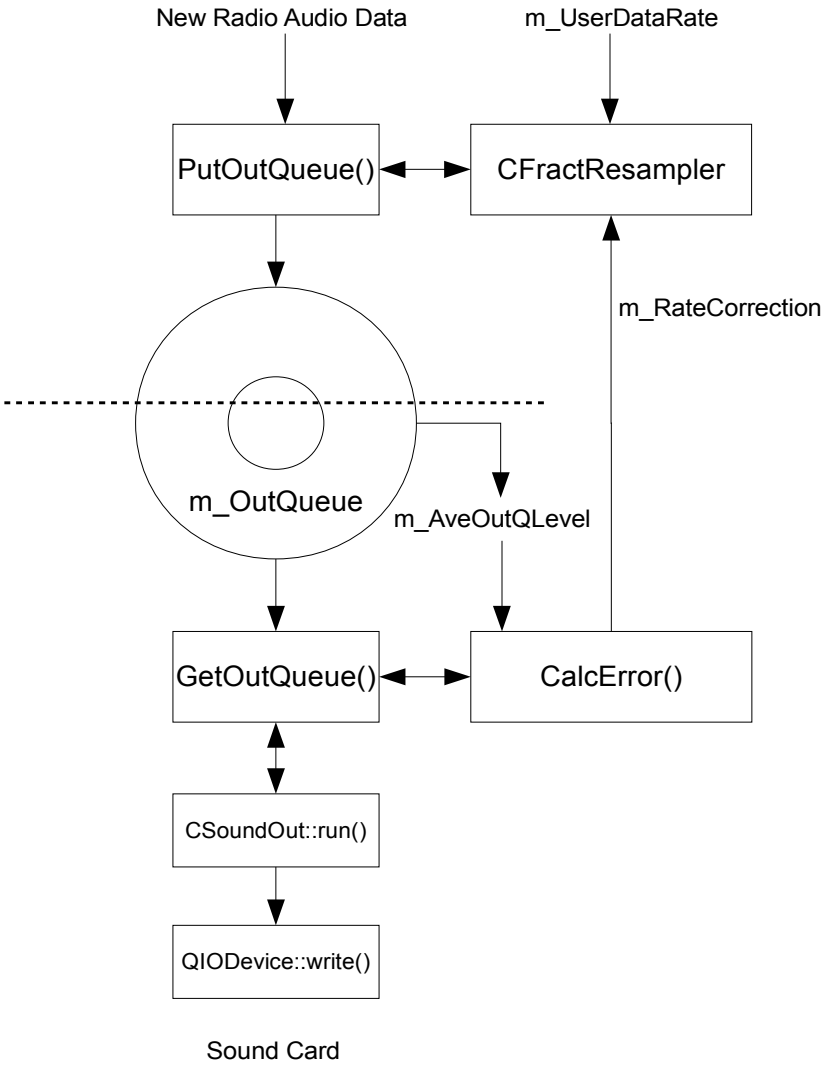
  

The CSoundOut::run() function in CSoundOut implements the worker thread loop used for polling and sending data to the QAudioOutput object.

A Data Queue is used between the sound card worker thread and the input thread that is sending data to the sound card. The input thread is the one that has just processed a block of data from the radio and is the CIQDataThread class that is pulling data from the network queue.

A secondary use of this queue is to monitor the sample rate differences between the sound card and radio data stream. By monitoring the filled level of the queue, one can determine the slight rate differences and feedback this rate error to try and keep the queue running without over or under flow.

Below shows the basic operation of the CSoundOut object.



The CFractResampler class discussed earlier is used to match the incoming radio audio sample rate to the fixed 48KHz sound card rate. An additional rate correction factor is applied by slowly averaging the data queue level and nudging the fractional rate adjustment towards the half-full mark.

Many algorithms were tried from PID control loops to complicated feedforward algorithms and as usually is the case, the simplest method worked the best. Basically a variable is kept that is incremented by the number of samples placed in the queue and subtracted by the number of samples removed from the queue. This variable is then averaged and then an error term is created from the average level to the ½ queue size. The error is multipield by a gain factor and the resulting rate correction value is applied to the fractional resampler to force the error term towards zero.

```
error = (double)(m_AveOutQLevel - OUTQSIZE/2 ); //neg==level is too low  pos == level is to high
error = error * P_GAIN;
m_RateCorrection = error;
```

In the PutOutQueue() function where new samples are placed in the data queue, the following call is made to the fractional resampler to create a new buffer of samples that are at the same rate of the soundcard. The TEST_ERROR value is just a debug constant to test verious rate error mismatches and see how the system responds.

```
//Call Resampler to match sample rates between radio and sound card
numsamples = m_OutResampler.Resample(numsamples, TEST_ERROR*m_OutRatio *(1.0+m_RateCorrection),
                                                    pData, RData, m_Gain);
```

A special blocking mode can be set when the sound card is started that bypasses the rate correction logic and also forces the input calling thread to wait for the sound card queue to have room for the new samples. This mode is not used by CuteSDR but could be used if the input source was from a file. The blocking mode would essentially gate the file reading rate to match the sound card rate otherwise the file reading rate would overflow the soundcard.

### *4.21. Noise Processing (CNoiseProc)*

Currently this class is more of a placeholder for future DSP processing blocks that are not essential for a basic radio but may be added at a later date. Various blocks that could be useful are noise blankers, noise reduction blocks, and auto notch filters.

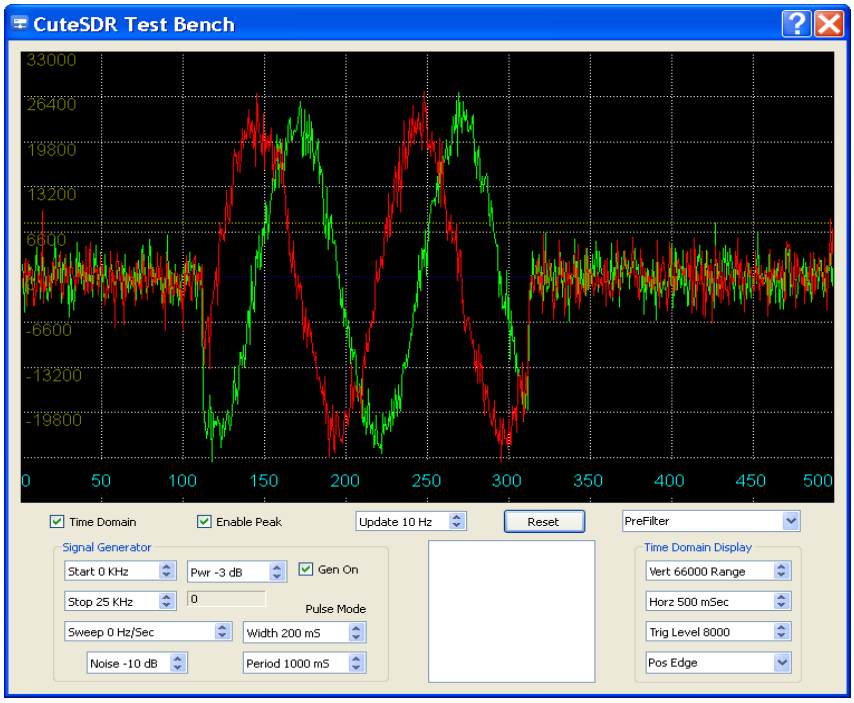
## *4.22. Test Bench Tool(CTestBench Class)*

During the course of CuteSDR development, it was found to be impossible to just use off the air signals or even RF signal generators to debug and verify all the various DSP modules. As a hardware engineer, it was desired to have the software equivalent of a signal generator, an oscilloscope, and spectrum analyzer that could probe various points in the DSP code to visually examine what was going on and make accurate measurements. It was also used in this document to provide graphical displays of various DSP components and their responses.

Thus the CTestBench class was started and grew as the project developed and more features were needed. This is a QDialog derived GUI class that contains several signal generation and display options. It is invoked as a non-modal pop-up window that can just run on the side almost as a separate program yet have intimate hooks into the CuteSDR code.

This class was not meant to be part of the final CuteSDR program but it was decided to leave it in as it may be of some use for those wishing to modify the code or just as a learning tool to examine various internal signals. It is not documented quite so well and was not written to be particularly efficient but since its use is temporary this is not a real issue as long as one realizes it is limited in performance and features.

The CTestbench screen looks somewhat like this(it tends to change depending on what was being tested). This is a pulsed complex carrier with Gausian noise added and displayed in the time domain.




The main screen can display either spectrum data or time domain data. There is an internal signal generator setup on the lower left and the time domain(Oscilloscope like) display controls on the lower right. Very limited triggering and range controls can be setup. The signal generator can output a single or swept complex carrier with adjustable amplitude in dB. The sweep generator is controlled by the start and stop frequencies and the sweep rate control. Setting the sweep rate to 0 makes the generator output the single start frequency. It can also output a rectangular complex carrier pulse with adjustable width and period. A Gaussian noise signal can also be added to the carrier.

### 4.22.1 Software Interface

The CTestBench object is instantiated using a global pointer, g_pTestBench. This way it can be used anywhere in the CuteSDR program without having to deal with passing pointers down into the nested depths of C++ objects.

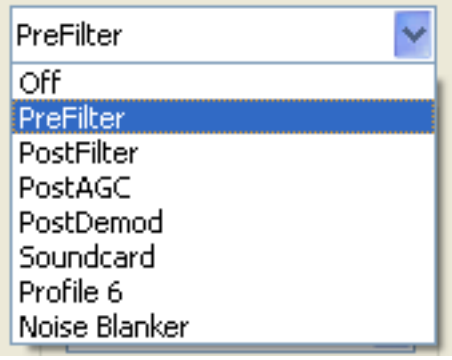
The generator function returns a buffer filled with samples from the generator.

An example call would be made like this to create 'Length' samples in user buffer pIQData, at a sample rate of m_SampleRate.

```
g_pTestBench->CreateGeneratorSamples(Length, pIQData, m_SampleRate);
```

In CuteSDR this is done at the beginning of the processing of the radio data and just substitutes the radio's data with its own. If the CTestBench object is inactive or the generator is turned off, then this function justr returns so can be left in line.

The display data for the CTestBench object has a similar function that can be placed at various points in the program. It has an additional profile parameter that allows selection of which function will supply the data to display. A drop down list of defined profiles in the GUI interface is used to select which function gets called.




The following example is how the data just after the main demod filter is selected for display by the Test Bench.

The constant “PROFILE_2” identifies the routine that will be active in collecting data to display. If “PostFilter” is selected in the GUI menu, then only the function with PROFILE_2 will be active.

```
//perform main bandpass filtering
n = m_FastFIR.ProcessData(n, m_pDemodInBuf, m_pDemodTmpBuf);
g_pTestBench->DisplayData(n, m_pDemodTmpBuf, m_OutputRate, PROFILE_2);
```

In CTestBench, the DisplayData function qualifies itself with the following code snippet:

```
void CTestBench::DisplayData(int length, TYPECPX* pBuf, double samplerate, int profile)
{
      if(!m_Active || (profile!=m_Profile) )
            return;
    .
    .
}
```

Various overloaded variants of the DisplayData() function are implemented for different data types.

The Data display can display either FFT spectrum data or Time domain data. An FFT peak hold mode will display a peak reading on a separate trace. The Reset button clears the peak and also re-starts the sweep generator if it is active.

### *4.23. CPU Performance Tool (Performance.cpp Module)*

A simple set of global C functions were implemented to be able to measure various processes in CuteSDR. It is based on reading a 64 bit counter that is in most Pentium class processors that just increments at the CPU clock rate.

There are lots of “gotchas” when using this counter but it can provide relative timing information if one averages the results and uses it within the same CPU thread.

Basically one places a Performance function call before and after a code segment that one wishes to measure the time it takes to execute. The Performance code simply reads the Pentium counter before and after and calculates the difference in CPU clock cycles, averages over a lot of samples and outputs the time to the debug screen in the Qt IDE.

The following function reads the 64 bit counter value using assembly instructions.

```
///////////////  Time measuring routine using Pentium timer
static quint64 QueryPerformanceCounter()
{
quint32 eax, edx;
quint64 val;
    __asm__ __volatile__("cpuid": : : "ax", "bx", "cx", "dx");
    __asm__ __volatile__("rdtsc":"=a"(eax), "=d"(edx));
    val = edx;
    val = val << 32;
    val += eax;
    return val;
}
```

InitPerformance() is called to initialize the average and maximum variables.

The functions StartPerformance() and StopPerformance() are called before and after the code segment to be measured.

ReadPerformance() is called by the user to display the time delta between the 2 calls. Average and maximum time values are displayed in seconds if one sets the constant CPUCLOCKRATEGHZ correctly to the CPU speed. Being calibrated is not neccesary if one only cares about relative times when optimizing code.

Typically one measures the time a code section takes to operate on a block of samples. If one tries to measure the time to process one sample, then the time it takes to call the Performance function will probably be longer and the results will be skewed. One can enter as a parameter to the Stop() routine the number of samples processed inbetween the Start-Stop calls. The ReadPerformance routine will then calculate the time per sample to process the entire block.

Because the operating systems are pre-emptive, there are no gaurantees that the CPU will not be interrupted while processing the code segment being measured. As a result it is best to average over many calls to get a more accurate reading.

# 5. Miscellaneous Issues

## *5.1. Bugs*

Several issues are pending with CuteSDR. One is an issue with the Qt UDP library. With high network data rates, it appears there is something that is chewing up CPU time somewhere in the bowels of Qt. Just reading UDP packets and not doing any processing causes a significant CPU load. This issue has been reported and verified as a bug by Qt.
http://bugreports.qt.nokia.com/browse/QTBUG-10587

The Qt graphics system also uses a lot of CPU time to perform fast display of waterfalls and other displays. This is probably not so much a bug but just a reality of having another layer of code wrapping the native GUI interface of the various supported platforms. This is just a fact of life unless one wants to bypass the Qt graphics and use the native API or perhaps use a 3rd party graphics library.

The Sound Card interface with Qt as described in a previous section leaves a lot to be desired. Not only the threading issue but the very large latency times are annoying at best. Perhaps playing with the buffer sizes can reduce the latency but at least with Windows, large buffers are needed to prevent drop outs.

Various CuteSDR bugs are sure to be found as the main focus has been on documentation and not on GUI or implementation bugs.

Not a lot of testing was done on MAC and Linux systems. It appeared that Windows was the worst platform in terms of network and soundcard interfacing so it was the primary development platform. The MAC GUI is somewhat odd for example the menu bar is not part of the program so Qt has some issues trying to utilize it. For example certain menu names must be reserved words or something and will not display on a MAC.
Only a Mint version of Linux was used for testing. It was assumed most Linux users would be able to grapple with any CuteSDR issues on other Linux variants and are used to recompiling the kernel or obtaining custom drivers etc.

## *5.2. Conclusions*

CuteSDR has proved to be a good learning experience. If the goal is to try and explain and document how a program works not just to make it run, then one is forced to understand more fully how things work instead of just writing code.

This is highly recommended if there is not a lot of time pressure to release code as the author will probably benefit more than any user ever will. This has certainly been the case with CuteSDR.

While the GUI interface is pretty much done since the goal was to keep it to a minimum, the DSP routines have lots of room for improvement, especially the AGC system. It is also desired to add various noise processing DSP functions in the future as well but it was decided to release the initial code without it.

The one DSP class that is not original is the FFT routines which were from a 3rd party. While quite useable, there were absolutely no comments or description in the code which goes against the very reason for CuteSDR. Writing an effficient FFT implemention is not a trivial undertaking and so was not attempted at this time. A secondary benefit of creating ones own FFT code would be to have a forward and reverse FFT function that does not re-order the terms in frequency order since it is not needed to do fast convolution.

# 6. References


J. Blanchette and M. Summerfield “C++ GUI Programming with Qt 4” Second Edition

A.V. Oppenheim and R. W. Schafer with J. Buck “Discrete-Time Singal Processing” Second Edition

M.E. Frerking “Digital Signal Processing in Communication Systems”

R. G. Lyons “Understanding Digital Signal Processing” Second Edition

P.M. Embryee and D. Danieli “C++ Algorithms for Digital Signal Processing” Second Edition

D.L. Jones “Fast Convolution”

P. Harman “A Discussion on the Automatic Gain Control(AGC) requirements of the SDR1000”

J.O. Smith “Digital Audio Resampling Home Page”, http://www-ccrma.stanford.edu/~jos/resample/, [20011-2-4].

Robert Bristow-Johnson. “Cookbook formulae for audio EQ biquad filter coefficients”

Freescale Semiconductor “Phase-Locked Loop Design Fundamentals” AN535

Haiyun Tang, “Notes on DPLL”

José A. Soares Augusto “The Io-sinh function, calculation of Kaiser windows and design of FIR filters”

L. E. Gugle ,“ S-Meter - Calibration & IARU Standards”

National Radio Systems Committee, “United States RBDS Standard”

Verigy “Introduction to FM-Stereo-RDS Modulation”

Lawrence Der Phd, Silicon Laboratories Inc., “Frequency Modulation(FM) Tutorial”

Michael Purser, “Introduction to Error-Correcting Codes”, ISBN 0-89006-784-8

C. Britton Rorabaugh, “Error Coding Cookbook”, ISBN 0-07-911720-1

http://www.google.com/ for all sorts of tidbits some good some not so good.


For information on RFSPACE Inc. radios -- www.rfspace.com