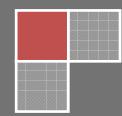


# The WDSP Guide

## Using WDSP - for Software Developers

WDSP is a full-featured signal processing library for Software Defined Radio. While assuming general SDR and programming knowledge, this guide provides the specifics of setting up and accessing WDSP.



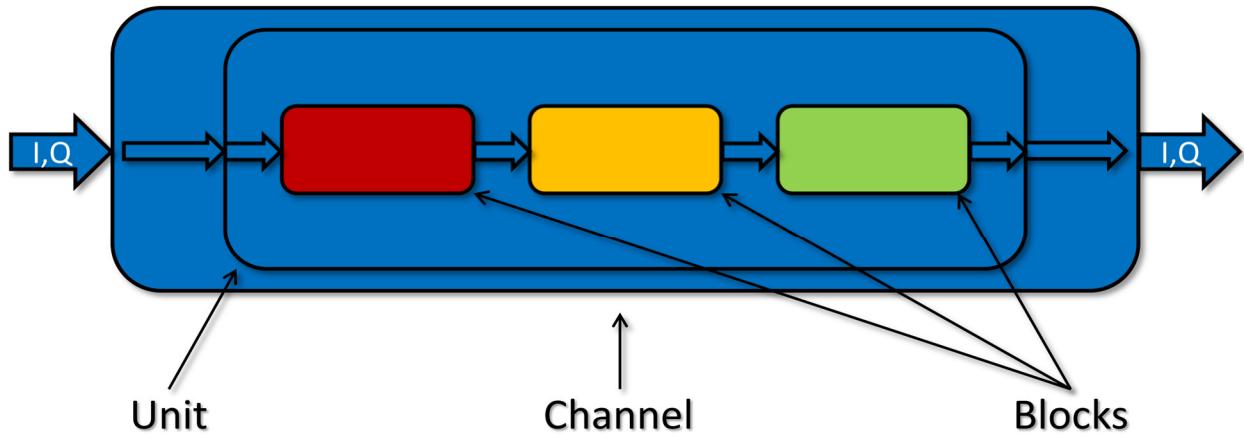
## Table of Contents

WDSP Organization .....	5
Operation of the Channel .....	6
Instantiation and Initialization .....	6
Setting the Channel State .....	7
Important Additional Information .....	8
Setting Other Channel Parameters .....	9
Exchanging Data.....	11
WDSP Wisdom Calculations.....	13
The RXA Receiver Unit .....	14
Channel Parameters for RXA.....	14
Signal Levels for RXA .....	15
Block Diagram & Controls .....	16
Frequency Shifter.....	17
Resamplers.....	18
Signal Generator .....	19
Input, S, and AGC Meters.....	21
Notched Bandpass Filter .....	22
Post-filter Display Sender.....	29
AM Squelch .....	30
AM/SAM Demodulator .....	31
Voice Squelch.....	32
FM Demodulator.....	33
FM Squelch.....	35
Spectral Noise Blanker .....	36
Equalizer.....	38
AGC.....	41
Automatic Notch Filter.....	48
LMS Noise Reduction .....	51
Spectral Noise Reduction.....	53
Bandpass Filter.....	58
Scope/Phase Display Sender.....	60

AM Carrier Block .....	61
CW Peaking Filter .....	62
Dolly Filter .....	66
Patch Panel - Audio Output Configuration .....	68
RXA Collectives & General Controls .....	70
The TXA Transmitter Unit .....	71
Channel Parameters for TXA .....	71
Signal Characteristics & Levels for TXA .....	72
Block Diagram & Controls .....	73
Resamplers .....	74
Signal Generators .....	75
PatchPanel .....	82
Meters .....	83
Noise Gate .....	84
Equalizer .....	85
FM Pre-emphasis .....	88
Leveler .....	90
Phase Rotator .....	91
Continuous Frequency Compressor (CFC) .....	93
Bandpass Filters .....	95
Speech Processor .....	99
CESSB Overshoot Control .....	100
ALC .....	101
AM Modulator .....	103
FM Modulator .....	104
Up-Slew .....	106
Siphon .....	107
PureSignal I/Q Predistortion .....	109
CFIR Filter .....	116
TXA Collectives & General Controls .....	117
Panadapter & Other Frequency-Domain Displays .....	118
Creating and Destroying a Display .....	119

Setting Display Parameters.....	120
Supplying Input Data.....	125
Retrieving Pixel Values.....	127
Clearing the Display Pipeline .....	127
Retrieving Maximum Bin Value (dB) Within a Frequency Range.....	127
Blocks Used Outside Channels.....	129
Preemptive Wideband Noise Blanker.....	129
Interpolating Wideband Noise Blanker.....	136
Resampler .....	143
Complex Double-Precision Resampler.....	143
Non-Complex (Audio) Single-Precision Resampler .....	145
Variable Ratio Resampler.....	148
Sample-Rate Matcher .....	152
Diversity Mixer .....	159
EER .....	163
Combined VOX & Downward Expander.....	166
Bandpass Filter Characterization Utility.....	174
Filter Impulse Response Cache .....	176
WDSP Version Information .....	177
Revision History .....	178

## WDSP Organization



Within WDSP, a “channel” is the home for a single “unit”. The channel provides input/output buffering to permit the unit to run asynchronously from the outside world and it provides to the unit a thread for its computations. When a channel is “opened”, it is told which type of unit, from a selection of types, it is to house. Channels are independent of each other. Each has its own fundamental parameters, e.g., buffer sizes and sample rates. Channels can operate in parallel, in series, or have no particular relationship at all to each other.

Units are generally major functional items such as a receiver, a transmitter, an interstellar exhorborator, or whatever other major signal processing one requires. However, should one need to wrap a single simple function within the channel structure, a unit could certainly be defined to do so.

The unit houses a number of blocks and buffers as needed to implement its function. For example, in the case of a transmitter, some of the blocks would be a compressor, a band-pass filter, and ALC.

The channel remains exactly the same no matter what unit is housed within. To function within a channel, there is, of course, a specific structure expected of the unit. Once written, the unit can be replicated in as many channels as desired. Likewise, there is a specific structure expected of the blocks that are housed within the units. This block structure supports multiple instances of any block within a unit; for example, you could write a block that implements a particular type of filter and then use three different instances of that filter with the same unit. Likewise, once written, a block can be instantiated within multiple types of units.

# Operation of the Channel

## Instantiation and Initialization

A channel is instantiated as the result of a call to [OpenChannel\(...\)](#).

```
void OpenChannel (int channel, int in_size, int dsp_size, int input_samplerate, int dsp_rate, int  
output_samplerate, int type, int state, double tdelayup, double tslewup, double tdelaydown, double  
tslewdown, int bfo)
```

The return type is **void** and the parameters are:

**int channel** – The unique identifier you wish to assign to this channel. Currently a number ranging 0 to 31. This range can be extended as needed.

**int in\_size** – The size (as a count of complex samples) that is to be provided as input to the channel each time an exchange function is called. This value is required to be a power of two.

**int dsp\_size** – The size (as a count of complex samples) that is used internally for DSP processing. This sets the size of buffers of data that are passed through the unit and therefore through its blocks. This parameter is required to be a power of two.

**int input\_samplerate** – The sample-rate of the input data to the channel.

**int dsp\_rate** - The sample rate used internally within the blocks of the unit for DSP processing. This rate may affect certain performance characteristics such as filter shape factors. It must be related to the channel input and output sample-rates by powers of two. It may either be a sub-multiple, equal, or a multiple; however, the ratio must be a power of two.

**int output\_samplerate** – The sample-rate of the output data from the channel (may or may not be the same as the input sample-rate). Note, however, that the output sample rate is required to be related to the input sample-rate by a power of two. It may either be a sub-multiple, equal, or a multiple; however, the ratio must be a power of two.

**int type** – A number that has been assigned to the particular type of unit that is to be housed within this particular channel. At this time, two channel types are available. For the RXA Receiver the type is 0 and for the TXA transmitter the type is 1.

**int state** - If set to 0 in [OpenChannel\(...\)](#), the channel will be created but will not operate until the state is later changed to 1; if set to 1 in [OpenChannel\(...\)](#), after creation, the channel will start, up-slew, and continue to operate as soon as it is provided with input data via an exchange function call.

**double tdelayup** - Time (in seconds) to delay before beginning an up-slew.

**double tslewup** - Length (in seconds) of an up-slew.

**double tdelaydown** - Time (in seconds) to delay before beginning a down-slew.

`double tslewdown` - Length (in seconds) of a down-slew.

`int bfo` - If non-zero, the call to an exchange function will NOT return until output data is available. This is useful in situations where input data is not guaranteed to arrive at regular intervals.

## Setting the Channel State

The `state` variable of the channel is changed using

`int SetChannelState (int channel, int state, int dmode)`

This function returns the PRIOR value of `state`, i.e., the value at the time the call was made.

Setting channel state is used anytime it is desired to turn ON or OFF the operation of a channel. (This is the ONLY call to turn ON/OFF a channel.) One common situation is the transmit-receive transition.

### Example 1: Receive to Transmit

For the receiver:           `SetChannelState (0, 0, 1);`

[Deactivate receiver hardware and fully activate transmit hardware here]

Then, for the transmitter:    `SetChannelState (5, 1, 0);`

The receiver will (1) wait `tdelaydown` before taking action, (2) down-slew its audio output during time `tslewdown`, (3) flush the channel and zero the output buffer, and (4) stop operating entirely so as to not use CPU cycles. With `dmode` set to 1, the function `SetChannelState(...)` will NOT return until these actions have been completed. Therefore, with the receiver and transmitter calls made in sequence, we know that the transmitter cannot begin its up-slew until the receiver is completely OFF. (More on `dmode == 0` later.) **Note that the receiver hardware should remain FULLY ACTIVE until the down-slew is complete; without samples flowing, it is impossible for the down-slew and a subsequent channel flush to occur.**

The transmitter will then (1) wait to see non-zero samples coming in, (2) wait `tdelayup` before taking action, (3) up-slew its MIC audio input during time `tslewup`, and (4) remain ON until its state is again changed. **Note that the transmitter hardware should be FULLY ACTIVE before the up-slew begins; otherwise, an abrupt turn-on in transmitter power (a discontinuity) would result.**

### Example 2: Transmit to Receive

For the transmitter:        `SetChannelState (5, 0, 1);`

[Deactivate transmitter hardware and fully activate receiver hardware here]

Then, for the receiver:     `SetChannelState (0, 1, 0);`

The transmitter will (1) wait `tdelaydown` before taking action, (2) down-slew its I-Q output during time `tslewdown`, (3) flush the channel and zero the output buffer, and (4) stop operating entirely so as to not

use CPU cycles. With `dmode` set to 1, the function `SetChannelState(...)` will NOT return until these actions have been completed. Therefore, with the transmitter and receiver calls made in sequence, we know that the receiver cannot begin its up-slew until the transmitter is completely OFF. (More on `dmode == 0` later.) **Note that the transmitter hardware should remain FULLY ACTIVE until the down-slew is complete; without samples flowing, it is impossible for the down-slew and a subsequent channel flush to occur.**

The receiver will then (1) wait to see non-zero samples coming in, (2) wait `tdelayup` before taking action, (3) up-slew its I-Q input during time `tslewup`, and (4) remain ON until its state is again changed. **Note that the receiver hardware should be FULLY ACTIVE before the up-slew begins; otherwise, an abrupt turn-on in received signal (a discontinuity) would result.**

#### Example 3: Receive to Transmit, when we want to turn off multiple receivers

The primary difference here is that if we were to turn-off all receivers with `dmode` set to 1, each would down-slew and turn-off in sequence. This would mean more delay than necessary in making the transition from receive to transmit. The receivers should slew down and turn-off in parallel.

For the first receiver:      `SetChannelState (0, 0, 0);`

For the second receiver:      `SetChannelState (1, 0, 1);`

[Deactivate receiver hardware and fully activate transmit hardware here]

For the transmitter:      `SetChannelState (5, 1, 0);`

In this case, the call for the first receiver will immediately return. The call for the second receiver, however, will wait until that receiver has turned-off. As long as the first receiver has the same or less delay and slew time specified, it should also be off when the call to the second receiver returns. Then, after flipping the hardware, the transmitter is turned on.

There could be very minor differences in execution times for the two receivers; however, those should be relatively insignificant compared to any delay and slew times. A very small transmitter delay can be used to resolve any differences.

#### **Important Additional Information**

- When you call `SetChannelState(...)` with `state == 1`, `dmode` has no effect. The call begins the turn-on sequence and returns immediately. (Exception: If you call to begin the turn-on sequence while a turn-off sequence is in process, the function waits until the turnoff completes, initiates the turn-on, then returns.)
- If you call `SetChannelState(...)` with `state == 1` when its state is already 1, nothing new happens; the function just returns. If an up-slew was in process, it just continues.
- If you call `SetChannelState(...)` with `state == 0` when its state is already 0, nothing new happens; the function just returns. If a down-slew was in process, it just continues. Also, note here that if `dmode` is set to 1 to "shadow" a prior call to turn OFF a receiver, there will be no delay in the

return of the call and the "shadowing" will not happen. You can check for this condition since the call returns the prior **state**.

- This function has a "Time-Out." If no input data is being provided to the channel, and this function is called to set **state** to 0, and **dmode** == 1, then, in the absence of a Time-Out, the function would never return because without data it can never complete the down-slew and subsequent flush. Similarly, if the function is called to set **state** to 1 and if a previous down-slew has not completed and has no data to complete, the function would never return. The Time-Out will cause the function to return after an extended time, currently set internally to ~200mS. This Time-Out feature is useful if the console is, for some reason, trying to manipulate the channel state in the absence of dataflow, for example during console start-up before data is flowing.

## Setting Other Channel Parameters

Primary channel parameters are those that are initially specified in the [OpenChannel\(...\)](#) call that creates the channel. These parameters can be reset to different values at later times using appropriate calls. Resetting the channel **state** has been discussed above. Resetting of other parameters is discussed here.

### **void SetDSPBuffsize (int channel, int dsp\_size)**

Sets the buffer size used for DSP internal processing. **dsp\_size** will determine the size of FFTs used for the bandpass filters and many other filters, and it will determine the size of buffers processed by other blocks. **dsp\_size** is required to be a power of two in order to achieve highest efficiency in FFT operation.

This function can be called at any time, **on the control thread**, including when data is flowing on the data input thread. It sets the channel **state** to 0, makes the necessary changes, and then sets the channel **state** back to its previous value. If data is flowing, it will necessarily pause channel processing to make the changes; however, it will not block the data input thread during this time. Data output will properly slew to zero, flush the channel, stay at zero during the changes, and properly slew back up to full strength.

**Data flow should NOT be terminated during the time this function is executing. Doing so could result in a "broken" down-slew and the lack of a channel flush.** This should be easily manageable when the flow of data is controlled on the same control thread that calls this function.

### **void SetDSPSamplerate (int channel, int dsp\_rate)**

Sets the sample rate used for internal DSP processing. Note that, among other things, **dsp\_rate** will affect the shape factor of filters.

This function can be called at any time, **on the control thread**, including when data is flowing on the data input thread. It sets the channel **state** to 0, makes the necessary changes, and then sets the channel **state** back to its previous value. If data is flowing, it will necessarily pause channel processing to make the changes; however, it will not block the data input thread during this time. Data output will

properly slew to zero, flush the channel, stay at zero during the changes, and properly slew back up to full strength.

**Data flow should NOT be terminated during the time this function is executing. Doing so could result in a "broken" down-slew and the lack of a channel flush.** This should be easily manageable when the flow of data is controlled on the same control thread that calls this function.

#### **void SetInputSamplerate (int channel, int in\_rate)**

The channel's **state** must be set to 0 (use **dmode == 1** if calling **SetChannelState(...)**) to be sure the operation is complete) before this function is called. It may be useful to make other changes that have a similar requirement, for example changing the channel's "Input Buffsize," while the channel **state** is still 0 and then returning it to 1 if appropriate.

#### **void SetInputBuffsize (int channel, int in\_size)**

The channel's **state** must be set to 0 (use **dmode == 1** if calling **SetChannelState(...)**) to be sure the operation is complete) before this function is called. It may be useful to make other changes that have a similar requirement, for example changing the channel's "Input Samplerate," while the channel **state** is still 0 and then returning it to 1 if appropriate.

#### **void SetOutputSamplerate (int channel, int out\_rate)**

The channel's **state** must be set to 0 (use **dmode == 1** if calling **SetChannelState(...)**) to be sure the operation is complete) before this function is called.

#### **void SetAllRates (int channel, int in\_rate, int dsp\_rate, int out\_rate)**

This is an alternative call that can be used to set all three channel rates at the same time. The channel's **state** must be set to 0 (use **dmode == 1** if calling **SetChannelState(...)**) to be sure the operation is complete) before this function is called.

#### **void SetChannelTDelayUp (int channel, double time)**

**channel:** identifier/number of the channel being accessed

**time:** time (seconds) to delay before beginning an up-slew

#### **void SetChannelTSlewUp (int channel, double time)**

**channel:** identifier/number of the channel being accessed

**time:** length of the up-slew (seconds)

#### **void SetChannelTDelayDown (int channel, double time)**

**channel:** identifier/number of the channel being accessed

**time**: time (seconds) to delay before beginning a down-slew

**void SetChannelTSlewDown (int channel, double time)**

**channel**: identifier/number of the channel being accessed

**time**: length of the down-slew (seconds)

**void SetType (int channel, int type)**

Changing the channel **type** requires completely destroying the old unit and building a new one. While the channel parameters as specified in the [OpenChannel\(...\)](#) call will be preserved, all parameters that had been stored in the blocks of the unit will be destroyed and the new unit will be created with default values.

## Exchanging Data

The channel uses a time-slice-exchange model meaning that, immediately after a channel consumes a set of samples representing the signal for a certain window of time, a set of samples representing that same amount of time will be written out in exchange. Note that the number of in-samples may be different than the number of out-samples if a sample-rate change occurred somewhere in the channel processing. For example, in the case of a receiver channel, the incoming I/Q sample rate might be 384K and the output audio sample rate might be 48K. Therefore, 1024 input samples would be exchanged for 128 output samples.

All exchanges are for a power-of-two quantity of complex samples. The input samples will be consumed just BEFORE the output samples are written out. Therefore, the same buffer can be used for input and output samples if desired. There is a choice of exchange calls, depending upon the organization of the data. For example, if your data is organized in buffers of datatype double with interleaved I and Q, you can call:

```
//double, interleaved I/Q
void fexchange0 (int channel, double* in, double* out, int* error)
```

You are providing the channel number, a pointer to the input buffer, a pointer to the output buffer (which could be the same as the input buffer), and a pointer to an error indicator. The error indicator will indicate whether the unit had provided data or whether zeros were returned instead. For example, suppose a unit isn't getting enough CPU cycles to keep up, in that case an error will be indicated and zeros will be returned. **\*error** will be a non-zero value in the event of an error and will otherwise be returned as 0.

As another example, suppose your data is stored in separate I and Q buffers and the datatype is float. Then you'd use:

```
//separate I/Q buffers
void fexchange2 (int channel, INREAL* lin, INREAL* Qin, OUTREAL* lout, OUTREAL* Qout, int* error)
```

INREAL and OUTREAL is normally defined as `float` in the file "comm.h".

When changing "fundamental channel parameters", calls to a 'fexchange' function may continue; however, no new data will be accepted until the reconfiguration is complete. The thread calling 'fexchange' will not be blocked.

Note that the `fexchange0(...)` function (interleaved, double, I and Q) is the preferred method of providing samples. Interleaved values is the standard for type 'complex' and the FFTW library uses this format. Using `fexchange2(...)` with its separate I and Q buffers also requires more CPU cycles since a conversion to interleaved format must be made.

## WDSP Wisdom Calculations

Several of the Blocks provided make use of the FFTW library for FFT operations. A function is provided to calculate the FFTW wisdom file.

```
void WDSPwisdom (char* directory)
```

'directory' is a pointer to a string containing the path to the directory where the wisdom file is to be stored.

This function should be called each time the application is opened. It will load the wisdom file if it is already present or create it if it is not.

The wisdom file includes all Forward/Backward, Complex/Real wisdom required for both RXA / TXA processing and for display processing.

A console program will report progress as the wisdom file is initially generated. In addition, a string containing information on the most recent progress in creating the wisdom file can be periodically read. This is accomplished with the call:

```
char* wisdom_get_status ()
```

A pointer to the string is returned.

## The RXA Receiver Unit

### Channel Parameters for RXA

The channel code serves as the interface between the unit it houses and the outside world. Considering this, some of the channel parameters relate only to the outside world. However, others impact the operation or perceived operation of the unit. It is this latter category of parameters that we discuss again here.

**dsp\_size:** The unit only processes data when `dsp_size` samples have been accumulated. This parameter therefore impacts the latency incurred in DSP processing. Generally, using a lower `dsp_size` will reduce latency. However, these lower values mean a larger number of buffers are processed per unit time and each buffer requires some amount of processing overhead. Therefore, lower `dsp_size` implies some increase in the CPU load. **For RXA, running at a `dsp_rate` of 48000, `dsp_size` ranging from a minimum of 64 to a maximum of 1024 offers a reasonable range of trade-offs -- 64 for minimum latency and 1024 to minimize CPU load.**

Note, however, that there is an important relationship among `dsp_size`, `dsp_rate`, `in_size`, and `input_samplerate`. For example, consider the case where `dsp_size` = 64, `dsp_rate` = 48000, `input_samplerate` = 192000 and `in_size` = 256. As discussed below, a resampler will be automatically invoked to resample the input from 192000 down to 48000. For each four samples at the 192000 input rate, only one sample at the 48000 rate will be produced. Therefore, 256 input samples (`in_size`) will be required to populate a buffer of 64 such that DSP processing can proceed. This implies that making `in_size` less than 256 will not speed-up DSP processing. In this example, making it 128 would only mean that input must be supplied twice before a single DSP buffer could be processed.

**dsp\_rate:** `dsp_rate` not only impacts latency, as described just above, it also impacts the results in filtering operations. Doubling `dsp_rate` doubles the width of filter transition bands. This can be precisely compensated by doubling the number of filter coefficients. (See later sections on band-pass filters.) Of course, increasing the rate and increasing the complexity of the filter, both require more CPU cycles. **RXA has been designed to run optimally at a `dsp_rate` of 48000 for modes other than FM and at a rate of 192000 for FM.**

**tdelayup, tslewup, tdelaydown, tslewdown:** Typical settings for these parameters are given here. Note that for non-CW modes they are more generous to give a very smooth sound while for CW they are more restricted for fast turnaround.

CW:            `tdelayup` = 0.005; `tslewup` = 0.010; `tdelaydown` = 0.000; `tslewdown` = 0.005.

Other modes: `tdelayup` = 0.010; `tslewup` = 0.025; `tdelaydown` = 0.000; `tslewdown` = 0.010.

## Signal Levels for RXA

RXA assumes that an input signal magnitude of 1.0 corresponds to 0dBm. Magnitude =  $\sqrt{I^2 + Q^2}$ .  
**It is recommended that the input be scaled to conform to this.**

Since all computations in WDSP are done in double-precision floating point, and assuming levels stay well within that range, most RXA blocks are level-independent. However, that is not exclusively true. There are a few places where absolute level matters:

- 1) Metering outputs. The meter outputs provide a 0 dBm output for a signal magnitude of 1.0. If your input scaling differs from this, you can compensate externally by adding or subtracting some fixed dBm value from the metering outputs.
- 2) Spectrum display (data following the band-pass filter). This output will need to be scaled accordingly if your input scaling is different than that prescribed above.
- 3) Squelch Thresholds. These will need to be scaled accordingly if your input scaling is different than that prescribed above.
- 4) AGC. Various AGC features such as "Slope" and "Hang" require specification of a maximum input level. By default, this is set to 1.0. If your maximum input level is different, you can change this value by calling:

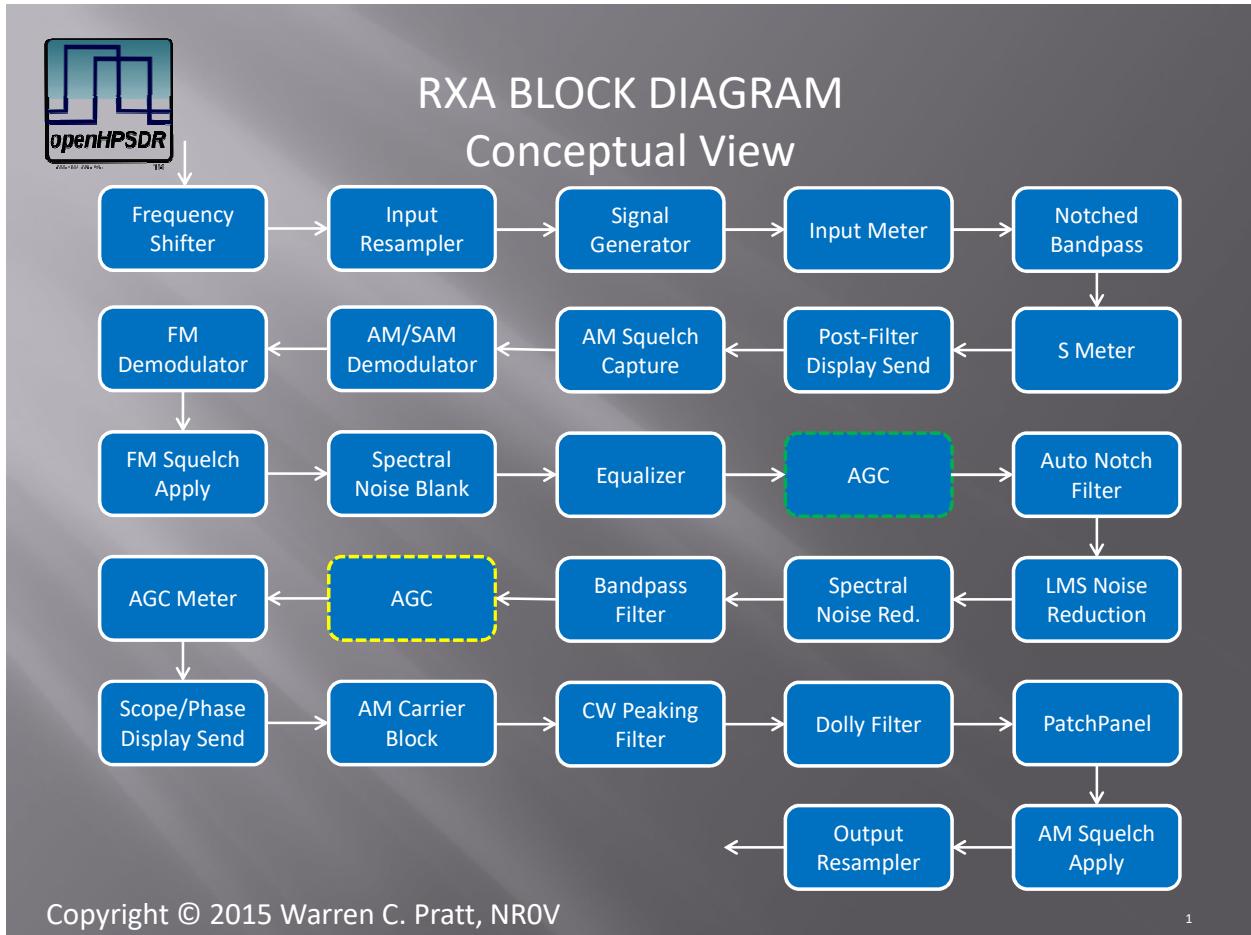
**void SetRXAAGCMaxInputLevel (int channel, double level)**

**channel**: number of the receiver channel being accessed

**level**: maximum input level.  
[default = 1.0]

## Block Diagram & Controls

A diagram showing the blocks comprising the RXA Unit is shown below. For clarity in explaining the operation, some minor simplifications have been made; therefore, this is titled as a "Conceptual View."



The purpose/function of each block and the calls to control each block will now be explained.

## Frequency Shifter

The frequency shifter generates a complex oscillator and multiplies the incoming sample stream by this oscillator. In analog radio terms, it's functions would be that of a "local oscillator" and "mixer." This process effectively rotates the incoming frequency spectrum by the frequency of the complex oscillator. There are several uses of this functionality, for example: (1) Click-Tune functionality where the panadapter display spectrum is static but the user clicks on a particular signal to receive, and (2) applying a final frequency offset at the output of the DFC (Direct Fourier Conversion) process.

Note that this is done BEFORE the resampler. Therefore, if resampling is to be done to a lower rate for internal DSP processing, a portion of the total input spectrum can be selected for further processing rather than only being able to rotate the portion of the spectrum available at the lower rate.

### `void SetRXAShiftRun (int channel, int run)`

**channel**: number of the receiver channel being accessed

**run**: turns the shifter OFF/ON, 0 for OFF, 1 for ON  
[default = 1]

### `void SetRXAShiftFreq (int channel, double fshift)`

**channel**: number of the receiver channel being accessed

**fshift**: frequency of the complex oscillator in Hertz  
[default = 0.0]

## Resamplers

In the discussion above of channel parameters, three distinct rates are mentioned:

**input\_samplerate**: samplerate of data entering the channel

**dsp\_rate**: internal rate at which the data is processed within the unit

**output\_samplerate**: samplerate of data exiting the channel

Two resamplers are provided within the RXA unit to convert among these rates. The first of these, the "Input Resampler," converts from the **input\_samplerate** to the **dsp\_rate**. The second of these, the "Output Resampler," converts from the **dsp\_rate** to the **output\_samplerate**.

No external calls are provided or needed to control these resamplers. The three rates are set by the channel parameters and the resamplers are automatically activated when rates differ and therefore resampling is required.

## Signal Generator

The signal generator is available to support testing and experimentation on RXA. When it is turned ON, it replaces the incoming samples with the signal samples from the generator output. It is not used for normal operation and is turned OFF by default. Therefore, its control calls can be completely ignored if the console does not desire to expose this functionality.

The signal generator can produce quite a few different types of signal outputs. However, only a subset of those have thus far been deemed useful for RXA and only that subset is currently supported with a full set of controls for RXA. The total set of possible outputs comprises: Tone, Two-Tone, Noise, Sweep, Sawtooth, Triangle, Pulse (can also pulse a tone on and off), and Silence. Sawtooth, Triangle, and Pulse produce an I-only (audio) output with Q = 0.0.

The set of outputs currently supported for RXA comprises: Tone, Noise, Sweep, and Silence.

**void SetRXAPreGenRun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 to turn the generator OFF, 1 to turn the generator ON  
[default = 0]

**void SetRXAPreGenMode (int channel, int mode)**

**channel:** number of the receiver channel being accessed

**mode:** Tone = 0, Noise = 2, Sweep = 3, Silence = 99  
[default = 2]

**void SetRXAPreGenToneMag (int channel, double mag)**

**channel:** number of the receiver channel being accessed

**mag:** peak magnitude value of the single-tone (mode = 0). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetRXAPreGenToneFreq (int channel, double freq)**

**channel:** number of the receiver channel being accessed

**freq:** frequency (Hertz) of the single-tone (mode = 0). Can be either positive or negative.  
[default = +1000.0]

**void SetRXAPreGenNoiseMag (int channel, double mag)**

**channel:** number of the receiver channel being accessed

**mag:** magnitude value of the noise (mode = 2). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetRXAPreGenSweepMag (int channel, double mag)**

channel: number of the receiver channel being accessed

mag: peak magnitude of swept single tone (mode = 3). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetRXAPreGenSweepFreq (int channel, double freq1, double freq2)**

channel: number of the receiver channel being accessed

freq1, freq2: frequency limits of the sweep range (Hertz). These may be either positive or negative. Sweep progresses from freq1 to freq2 and then repeats.  
[defaults: freq1 = -20000.0, freq2 = +20000.0]

**void SetRXAPreGenSweepRate (int channel, double rate)**

channel: number of the receiver channel being accessed

rate: rate at which the sweep progresses (Hertz / Second).  
[default = +4000.0]

## Input, S, and AGC Meters

As shown in the block diagram, three sets of RXA metering data are available to the console: Input-level, S-level (after the bandpass and notch filtering), and AGC-level (after the AGC). For Input, Input\_peak and Input\_average are available. For S, S\_peak and S\_average are available. For AGC, AGC\_peak, AGC\_average, and AGC\_gain are available.

The console can call for individual values any time it wants to update the respective meter user interfaces.

**double GetRXAMeter (int channel, int mt)**

**return value:** requested meter value (dBm, except dB for AGC\_gain)

**channel:** number of the receiver channel being accessed

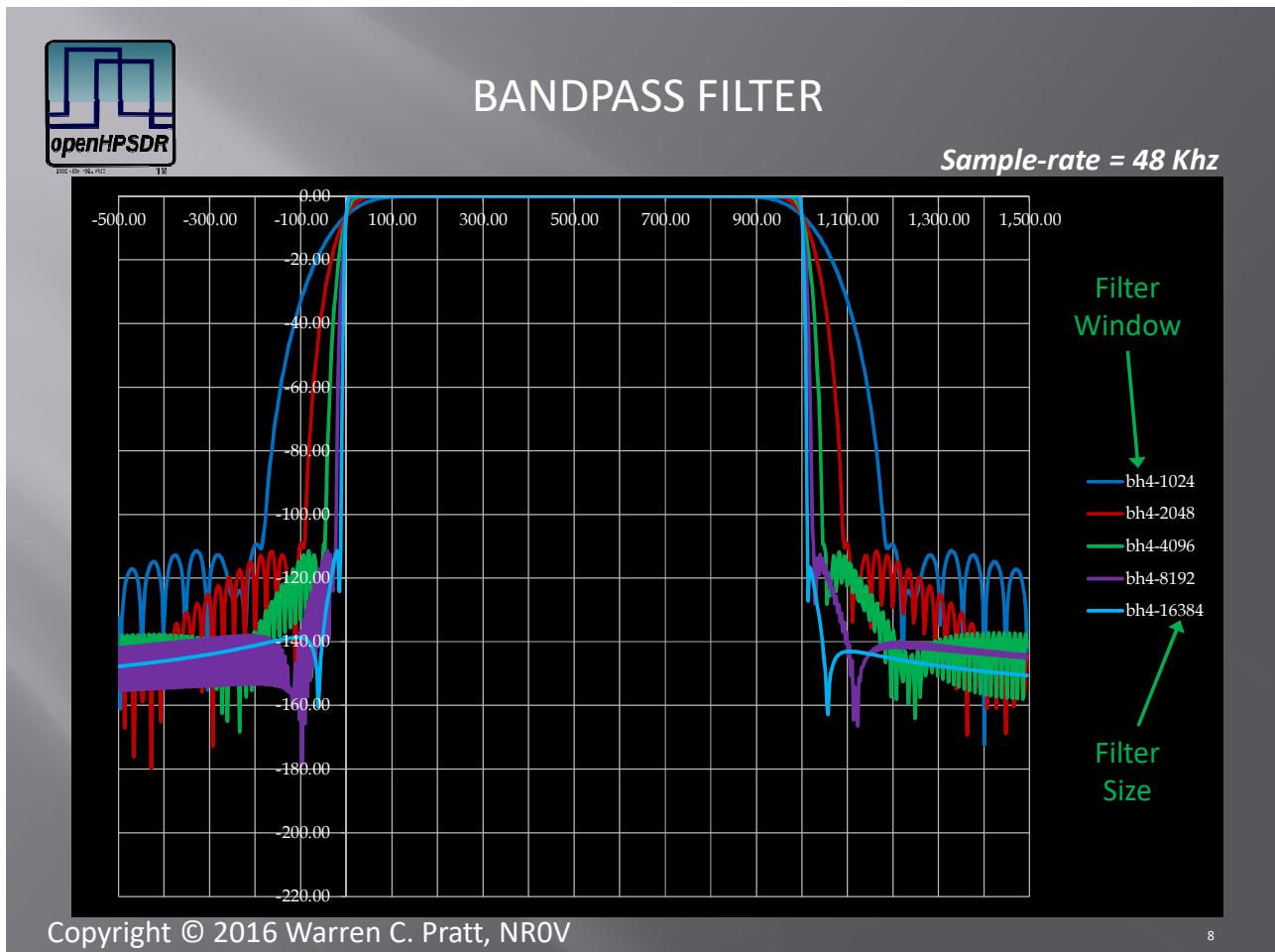
**mt:** type of meter data being requested. **mt** values are: S\_peak = 0, S\_average = 1, Input\_peak = 2, Input\_average = 3, AGC\_gain = 4, AGC\_peak = 5, AGC\_average = 6.

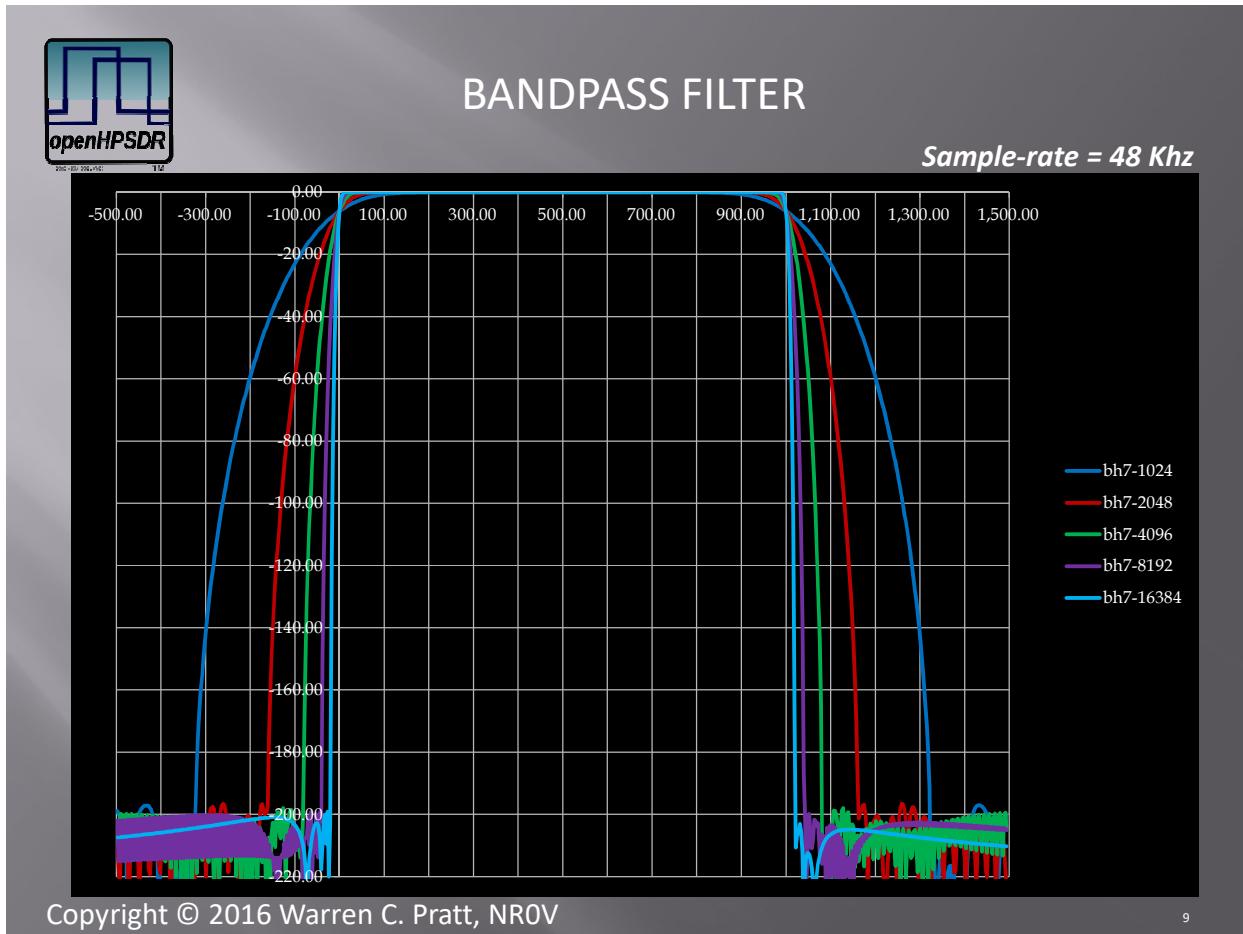
## Notched Bandpass Filter

The Notched Bandpass Filter provides both the primary bandpass filtering AND the fixed notch filtering (pre-specified notch frequencies as opposed to automatic notch filtering). Therefore, both the lower and upper bandpass limits and all the notch filtering information must be specified.

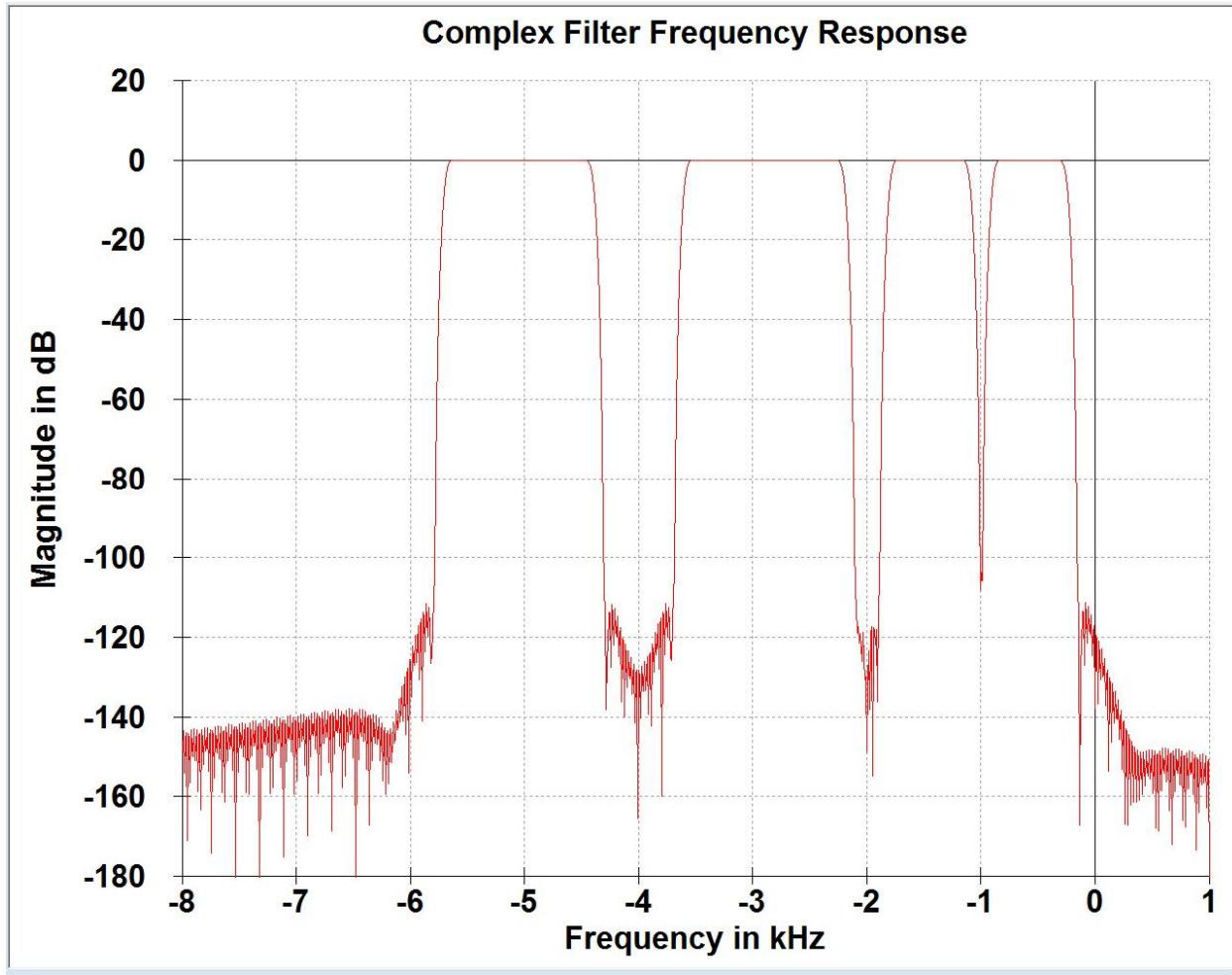
The "sharpness" of these filters, when running RXA at a `dsp_rate` of 48000, is shown in the following two figures. The first of these figures shows the response curves for different impulse response lengths, all using the 4-term Blackman-Harris window. The second figure is the same except using the 7-term Blackman-Harris window. The 4-term yields slightly sharper cutoff; however, less cutoff depth. The 4-term would be preferred in most situations since its cutoff depth approaches the dynamic range of the receiver hardware; the 7-term is overkill on cutoff depth for most situations.

Note that the transition bands (regions between the pass-band and the cutoff bands) are the most important part to observe in these figures. These transition bands will maintain the same width independent of the width of the filter pass-band.





The following figure displays an example of a Notched Bandpass Filter with three notches of three different widths. This example uses a filter size of 2048 at a `dsp_rate` of 48000. The notch at -1 KHz relative to the receiver tune frequency is 200 Hz in width, the notch at -2 KHz is 400 Hz in width, and the notch at -4 KHz is 800 Hz in width. Note that the width specifies the -6 dB points.



### *Bandpass Filter Settings*

**void RXANBPSetRun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 to turn the filter OFF; 1 to turn the filter ON.

[default = 1]

The lower and upper bandpass limits *can* be set with the **DEPRECATED** call:

**void RXANBPSetFreqs (int channel, double f\_low, double f\_high)**

**DEPRECATED**

**channel:** number of the receiver channel being accessed

**f\_low:** lower of the two frequencies (Hertz). May be negative or positive. This is a -6 dB point of the filter.

[default = -4150.0]

**f\_high**: higher of the two frequencies (Hertz). May be negative or positive. This is a -6 dB point of the filter.

[default = -150.0]

The preferred way to set the lower and upper bandpass limits is to use a "Collective" call which not only sets this filter properly but also sets the bandpass limits for other blocks that need that same information. See the section of this document on "RXA Collectives" for this preferred call.

#### **void RXANBPSetWindow (int channel, int wintype)**

**channel**: number of the receiver channel being accessed

**wintype**: 0 for 4-term Blackman-Harris window; 1 for 7-term Blackman-Harris window.

[default = 0]

The length of the desired filter impulse response *can* be set with the following call. This is NOT the preferred method. Instead, a "Collective" call can be used. See the section on "RXA Collectives."

#### **void RXANBPSetNC (int channel, int nc)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**nc**: length of the desired filter impulse response. This must be a power of two AND must be greater than or equal to the DSP Buffer Size in use.

[default = 2048]

The Filter-Type (Linear Phase or Low Latency) for the Notched Bandpass Filter can be set using the following call. This is NOT the preferred method. Instead, a "Collective" call can be used. See the section on "RXA Collectives."

#### **void RXANBPSetMP (int channel, int mp)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**mp**: 0 for Linear Phase; 1 for Low Latency (Minimum Phase)

[default = 0]

#### **Notch Filter Settings**

#### **void RXANBPSetNotchesRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 to turn OFF all specified notches; 1 to turn ON all specified notches that are 'active'

[default = 0]

The width of each notch is specified and implies the frequencies at the edges of the notch where the response is 6dB down. The transition bands of notches are not perfectly vertical; the transition bands

have some non-zero width. Therefore, if essentially full-cutoff (>100dB) is to be attained at the center of a notch, the notch must be specified with some minimum width. The value of this minimum width can be found using the call:

```
void RXANBPGetMinNotchWidth (int channel, double* minwidth)
```

**channel**: number of the receiver channel being accessed

**minwidth**: pointer to the location into which the minimum notch width is to be stored. The value stored is in Hertz.

The width of all notches will be automatically increased, when necessary, to that required to achieve >100dB cutoff at the notches' centers if the "autoincr" flag is set.

```
void RXANBPSetAutoIncrease (int channel, int autoincr)
```

**channel**: number of the receiver channel being accessed

**autoincr**: 0 sets Auto-Increase to OFF; 1 sets Auto-Increase to ON  
[default = 1]

Notches are specified at absolute RF frequencies. For example, if there is always an interfering carrier on 145.275 Mhz, a notch of a desired width may be specified at that frequency. The Notched Bandpass Filter then compares each notch frequency with the current RF frequency to which the radio is tuned to determine which notches are within the passband and where within the passband they are placed. To perform this comparison, the current radio Tune Frequency and any Shift Frequency offset to that must be specified. The following two calls are used for that purpose.

```
void RXANBPSetTuneFrequency (int channel, double tunefreq)
```

**channel**: number of the receiver channel being accessed

**tunefreq**: current RF tune frequency of the radio (Hertz)  
[default = 0.0]

```
void RXANBPSetShiftFrequency (int channel, double shift)
```

**channel**: number of the receiver channel being accessed

**shift**: current RF shift frequency (Hertz)  
[default = 0.0]

The Notched Bandpass Filter maintains a database of the desired notches. Several calls are provided to manage this database. Note that RXA currently supports a maximum of 1024 notches.

The following call can be made to determine the number of notches currently stored in the database.

```
void RXANBPGetNumNotches (int channel, int* nnotches)
```

**channel**: number of the receiver channel being accessed

**nnotches**: pointer to location where the number of notches currently in the database is to be stored

A new notch is added to the database using:

**int RXANBPAAddNotch (int channel, int notch, double fcenter, double fwidth, int active)**

**return value**: 0 if the operation was successful; -1 if the specified value of 'notch' was  $\geq$  the total (new, including this addition) number of notches, i.e., it was out-of-range, OR, if adding another notch would exceed the maximum number of notches that are supported.

**channel**: number of the receiver channel being accessed

**notch**: identifier for the notch to be added. Notch identifiers begin at 0 and are positive. Notches in the database are required to be consecutive, i.e., with no gaps in used notch identifiers. Therefore, the value of 'notch' must be less than the total (new, including this addition) number of notches. If 'notch' is equal to new\_number\_of\_notches minus one, this notch will be added at the end of the list. If 'notch' is less than new\_number\_of\_notches minus one, this notch will be added with the 'notch' identifier specified and the notches with higher identifiers will be pushed-up in the list and their identifiers incremented accordingly.

**fcenter**: center frequency of the notch (Hertz)

**fwidth**: width of the notch (Hertz)

**active**: 0 implies that even though the notch will be stored in the database, it will not be used in filtering operations. 1 implies that it will be stored and will be used in filtering operations.

The current parameters of a specified notch can be obtained using:

**int RXANBPGetNotch (int channel, int notch, double\* fcenter, double\* fwidth, int\* active)**

**return value**: 0 if the operation was successful; -1 if the value of 'notch' was  $\geq$  the current number of notches in the database.

**channel**: number of the receiver channel being accessed

**notch**: identifier for the notch to be read

**fcenter**: pointer to location to store center value of this notch (Hertz)

**fwidth**: pointer to location to store width of this notch (Hertz)

**active**: pointer to location to store the active flag for this notch

A specified notch can be deleted from the database using the following call. Note that any notches with higher identifiers will be pushed down in the list and their identifiers decremented accordingly.

**int RXANBDeleteNotch (int channel, int notch)**

**return value:** 0 if the operation was successful; -1 if the value of 'notch' was  $\geq$  the current number of notches in the database.

**channel:** number of the receiver channel being accessed

**notch:** identifier for the notch to be deleted

One or more parameters of a specified notch can be changed using the following call. Note that ALL the parameters must be supplied even though perhaps only some of them are being changed in value.

**int RXANBPEditNotch (int channel, int notch, double fcenter, double fwidth, int active)**

**return value:** 0 if the operation was successful; -1 if the value of 'notch' was  $\geq$  the current number of notches in the database.

**channel:** number of the receiver channel being accessed

**notch:** identifier for the notch to be edited

**fcenter:** center frequency of the notch (Hertz)

**fwidth:** width of the notch (Hertz)

**active:** 0 implies that even though the notch will be stored in the database, it will not be used in filtering operations. 1 implies that it will be stored and will be used in filtering operations.

## Post-filter Display Sender

The Post-filter Display Sender is used to transfer samples of the signal, taken after the Notched Bandpass Filter, to a WDSP Display Unit. This can provide a frequency-domain display showing the signal(s) within the passband and displaying the operation of the Notched Bandpass Filter and any other active preceding blocks in RXA.

Note that Display Units are instantiated and configured separately. See the appropriate section of this document for additional information on Display Units.

A single Display Unit may be used to process multiple data streams at different times, e.g., a full-bandwidth panadapter stream at some times as well as this post-filter data stream at others. Therefore, a 'flag' to turn the sender OFF/ON is needed.

Parameters are also needed to specify the target Display Unit, the target sub-span (in the event the display unit is set up for stitched displays), and which LO-position (in the event that spur-elimination mode is being used).

All this information is supplied in a single call:

**void SetRXASpectrum (int channel, int flag, int disp, int ss, int LO)**

**channel**: number of the receiver channel being accessed

**flag**: 0 to turn the sender OFF, 1 to turn the sender ON  
[default = 0]

**disp**: identifier of the display unit. The overall configuration *might* be such that there is one display unit per DSP channel and the number is the same as the channel number. However, this is not necessarily the case.

**ss**: the sub-span which is to receive the data, normally this will be set to 0.

**LO**: the LO position to receive the data, normally this will be set to 0.

## **AM Squelch**

The AM Squelch is provided for modes OTHER THAN FM. Special features of the WDSP implementation include raised-cosine OFF-ON-OFF transitions and a variable tail-length which depends upon the strength of the signal. Weaker signals get a longer tail-length to attempt to avoid the squelch turning the audio OFF/ON as the strength slightly varies.

### **void SetRXAAMSQRun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 for AM Squelch OFF; 1 for AM Squelch ON

### **void RXAAMSQThreshold (int channel, double threshold)**

**channel:** number of the receiver channel being accessed

**threshold:** squelch threshold, i.e., if the passband signal strength is below this level the audio will be squelched (turned-off). 'threshold' is a negative dBm value.

[default = -40.0]

### **void SetRXAAMSQMaxTail (int channel, double tail)**

**channel:** number of the receiver channel being accessed

**tail:** maximum tail length to be applied (seconds)

[default = 1.50]

## **AM/SAM Demodulator**

This demodulator provides either AM demodulation or Synchronous-AM demodulation, dependent upon settings. Note that the **SetRXAMode( ... )** collective makes the choice of AM or SAM, dependent upon the overall RXA Mode setting. (See the "RXA Collectives" section of this document.)

The following call exists to turn OFF/ON this demodulator; however, using this call is **NOT RECOMMENDED** as the demodulator is normally turned OFF/ON by using the **SetRXAMode( ... )** collective.

**void SetRXAAMDRun (int channel, int run)    DEPRECATED**

**channel:** number of the receiver channel being accessed

**run:** 0 to turn OFF the demodulator; 1 to turn ON the demodulator

For SAM ONLY, a choice of whether to receive both sidebands or only the lower or the upper sideband can be made. This is often useful in avoiding interference on one side or the other. Note that this is NOT a bandpass filtering operation; therefore, the received station does not have to be exactly centered on frequency for the sideband selection to function. This is particularly useful in AM round-tables where not all stations are on exactly the same frequency and interference exists on one side.

**void SetRXAAMDSBMode (int channel, int sbmode)**

**channel:** number of the receiver channel being accessed

**sbmode:** 0 = receive both sidebands; 1 = receive LSB only; 2 = receive USB only  
[default = 0]

For both AM and SAM demodulation, there is an option to "level" the carrier in situations of carrier fading. This is accomplished by stripping out the varying carrier and replacing it with a stable carrier. This feature is deactivated/activated with the following call.

**void SetRXAAMDFadeLevel (int channel, int levcfade)**

**channel:** number of the receiver channel being accessed

**levcfade:** 0 = OFF; 1 = ON  
[default = 1]

## Voice Squelch

Voice Squelch is offered as an alternative to the volume-based AM Squelch. It operates by attempting to distinguish a voice signal from noise and various types of interference.

Four controls are provided.

### **void SetRXASSQLRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 for Voice Squelch OFF; 1 for Voice Squelch ON

### **void RXASSQLThreshold (int channel, double threshold)**

**channel**: number of the receiver channel being accessed

**threshold**: squelch threshold. The value should be between 0.0 and 1.0. A typical value would be 0.16.

### **void RXASSQLTauMute (int channel, double tau\_mute)**

**channel**: number of the receiver channel being accessed

**tau\_mute**: time constant (seconds) to be used in muting the squelch. A reasonable (but wide) range would be 0.1 to 2.0. Typical values are near the low-end of the range.

### **void RXASSQLTauUnMute (int channel, double tau\_unmute)**

**channel**: number of the receiver channel being accessed

**tau\_unmute**: time constant (seconds) to be used in un-muting the squelch. A reasonable (but wide) range would be 0.1 to 1.0. Typical values are near the low-end of the range.

## FM Demodulator

The FM Demodulator supports the standard FM audio bandwidth of 300 Hz to 3000 Hz and applies the standard de-emphasis of 6 dB / octave.

The FM demodulator is turned OFF/ON by using the **SetRXAMode( ... )** collective. Hence, no call is provided here to accomplish that function.

The expected maximum deviation of the received signal is set as follows:

**void SetRXAFMDeviation (int channel, double deviation)**

**channel**: number of the receiver channel being accessed

**deviation**: deviation in Hertz

FM signals may have CTCSS tones used to trigger repeaters. Those may be passed through by the repeaters and it is sometimes desirable to notch those out of the audio output. A special notch filter is provided to perform this task. Two controls are provided for this filter:

**void SetRXACTCSSRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 = turn OFF the CTCSS notch filter; 1 = turn ON the CTCSS notch filter

**void SetRXACTCSSFreq (int channel, double freq)**

**channel**: number of the receiver channel being accessed

**freq**: frequency of the tone and notch (Hertz)

There are two filters (de-emphasis and audio) used in the FM demodulator and, for each of them, the length of the impulse response and the selection of Linear-Phase/Low-latency can be set. However, this is **NOT RECOMMENDED** and it is preferred that this be handled by calls to the RXA Collectives **RXASetNC(...)** and **RXASetMP(...)**. (See the "RXA Collectives" section of this document.)

**void SetRXAFMNCde (int channel, int nc)** DEPRECATED

**channel**: number of the receiver channel being accessed

**nc**: length of the impulse response. This must be a power of two and must be greater than or equal to the **dsp\_size** currently in use.

**void SetRXAFMMMPde (int channel, int mp)** DEPRECATED

**channel**: number of the receiver channel being accessed

**mp**: 0 = Linear Phase; 1 = Low Latency

`void SetRXAFMNCaud (int channel, int nc)`

**DEPRECATED**

`channel`: number of the receiver channel being accessed

`nc`: length of the impulse response. This must be a power of two and must be greater than or equal to the `dsp_size` currently in use.

`void SetRXAFMMPaud (int channel, int mp)`

**DEPRECATED**

`channel`: number of the receiver channel being accessed

`mp`: 0 = Linear Phase; 1 = Low Latency

The FM demodulator also includes a “volume limiter” that can be used to level the noise and signal volumes during the squelch tail or when squelch is OFF to listen for very weak signals. There are two controls for this functionality.

`void SetRXAFMLimRun (int channel, int run)`

`channel`: number of the receiver channel being accessed

`run`: 0 = turn OFF the volume limiter; 1 = turn ON the volume limiter

`void SetRXAFMLimGain (int channel, double gaindB)`

`channel`: number of the receiver channel being accessed

`gaindB`: the gain in dB that the limiter should apply. A reasonable range is 0dB to 30dB.

The normal frequency range for narrow-band FM communication is 300Hz to 3000Hz. This can be adjusted, if desired, to offer higher fidelity. However, note that going lower means that the receive bandwidth overlaps the range in which CTCSS tones are transmitted. Going higher in frequency increases the necessary receive bandwidth which, per Carson’s Rule, is  $2 * (\text{peak\_deviation} + \text{highest audio frequency})$ .

`void SetRXAFMAFFilter (int channel, double low, double high)`

`channel`: number of the receiver channel being accessed

`low`: low-cut frequency (Hz)

`high`: high-cut frequency (Hz)

## FM Squelch

FM Squelch is turned OFF/ON using the following call:

**void SetRXAFMSQRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 = turn OFF the FM Squelch; 1 = turn ON the FM Squelch  
[default = 0]

**void SetRXAFMSQThreshold (int channel, double threshold)**

**channel**: number of the receiver channel being accessed

**threshold**: squelch threshold. This is a positive value which normally ranges from 1.0 to 0.01.  
It can be generated with a slider from which the output value range is 0.0 to 100.0 and then  
applying the equation:  $\text{threshold} = \text{pow}(10.0, -2.0 * \text{slider\_value} / 100.0);$

A filter is used in the FM squelch and the length of the impulse response and the selection of Linear-Phase/Low-latency can be set. However, this is **NOT RECOMMENDED** and it is preferred that this be handled by calls to the RXA Collectives **RXASetNC (...)** and **RXASetMP (...)**. (See the "RXA Collectives" section of this document.) The calls to individually set these values are:

**void SetRXAFMSQNC (int channel, int nc)      DEPRECATED**

**channel**: number of the receiver channel being accessed

**nc**: length of the impulse response. This must be a power of two.

**void SetRXAFMSQMP (int channel, int mp)      DEPRECATED**

**channel**: number of the receiver channel being accessed

**mp**: 0 = Linear Phase; 1 = Low Latency

## Spectral Noise Blanker

The Spectral Noise Blanker uses Linear Predictive Coding (LPC) to detect impulses or other severe abnormalities in signals and to replace the corrupted samples with an estimate of the original signal.

NEW! 2015 SPECTRAL NOISE BLANKER  
“SNB”

NR	ANF
NB	SNB
MUT	BIN
MNF	

- Linear Predictive Coding (LPC) for detection and correction
  - $\hat{y}(m) = \sum a_k y(m - k)$
  - $a_k$  contain information about the Spectral Formants of the signal
- Detection:
  - Compare:  $Error(m) = y(m) - \hat{y}(m)$
  - Normally we find a sequence of corrupt samples
- Correction:
  - Using coefficients  $a_k$ , solve a system of linear equations for the values of the samples in the corrupt sequence

Copyright © 2016 Warren C. Pratt, NROV

The Spectral Noise Blanker is turned OFF/ON by calls to:

**void SetRXASNBARun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 for OFF; 1 for ON

[default = 0]

The output bandwidth of the Spectral Noise Blanker can be set to be consistent with the overall bandpass lower and upper limits using the following **DEPRECATED** call. This is NOT recommended. Instead, use the RXA Collective **RXASetPassband ( ... )** which sets all blocks that have need of passband frequency information. (See the "RXA Collectives" section of this document.)

**void SetRXASNBAOutputBandwidth (int channel, double flow, double fhigh) DEPRECATED**

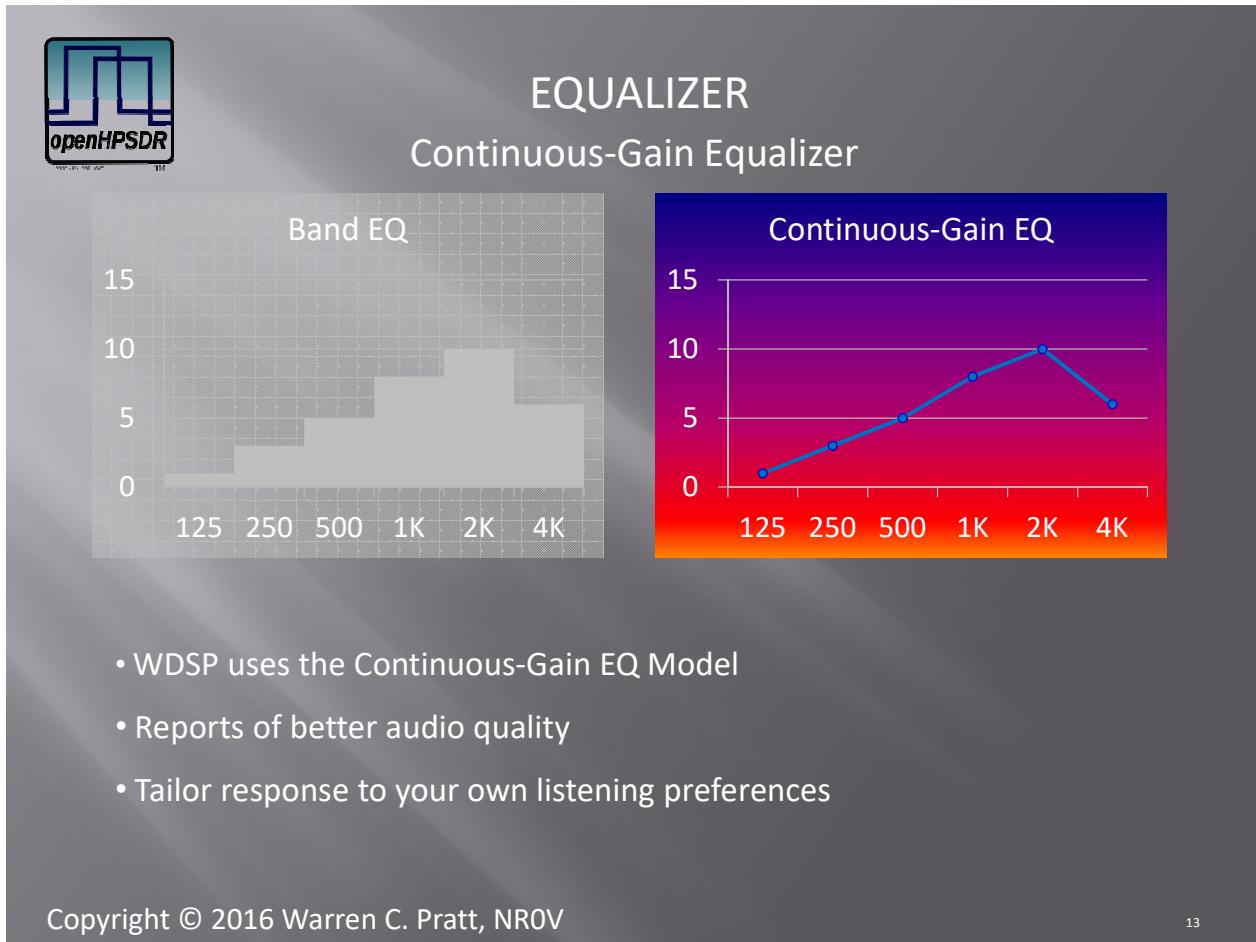
**channel**: number of the receiver channel being accessed

**flow**: lower bandpass limit (Hertz)

**fhigh**: upper bandpass limit (Hertz)

## Equalizer

The equalizer uses a continuous frequency response algorithm meaning that the gain is specified at any number of frequencies and the response is interpolated between the specified points. This is in contrast to a frequency "band" approach.



The equalizer is turned OFF/ON using calls to:

**void SetRXAEQRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 for OFF; 1 for ON  
[default = 0]

The length of the EQ filter impulse response can be set with the following call. However, this is **NOT RECOMMENDED**. The preferred method is to use the RXA Collective **RXASetNC(...)**. (See the "RXA Collectives" section of this document.)

**void SetRXAEQNC (int channel, int nc)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**nc**: length of the EQ filter impulse response. 'nc' must be a power-of-two and must be greater than or equal to the DSP Buffer Size in use.

[default = 2048]

The Filter Type (Linear-Phase / Low-Latency) for the EQ filter can be set with the following call. However, this is **NOT RECOMMENDED**. The preferred method is to use the RXA Collective **RXASetMP(...)**. (See the "RXA Collectives" section of this document.)

**void SetRXAEQMP (int channel, int mp)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**mp**: 0 for Linear-Phase; 1 for Low-Latency (Minimum Phase)

[default = 0]

The window-type used in the generation of the EQ filter can be selected. This setting should normally just be left at the default value.

**void SetRXAEQWinType (int channel, int wintype)**

**channel**: number of the receiver channel being accessed

**wintype**: 0 = 4-term Blackman-Harris; 1 = 7-term Blackman-Harris

[default = 0]

The cutoff-mode used in the generation of the EQ filter can be selected. This setting should normally just be left at the default value.

**void SetRXAEQCtfmode (int channel, int mode)**

**channel**: number of the receiver channel being accessed

**mode**: 0 = 4<sup>th</sup>-power roll-off on each side of the specified passband; 1 = no roll-off outside the specified passband

[default = 0]

The following call is the primary and preferred method of setting the equalizer response.

**void SetRXAEQProfile (int channel, int nfreqs, double\* F, double\* G)**

**channel**: number of the receiver channel being accessed

**nfreqs**: number of frequency/gain pairs to be specified

**F**: pointer to an array of frequencies (Hertz). NOTE: The 0<sup>th</sup> element of this vector is NOT USED. So, for example, if **nfreqs** is 5, F[0] is not used and the five frequencies will be stored in F[1] ... F[5]. The F[] array has (**nfreqs** + 1) values.

Note that acceptable values are between 0.0 Hz and **dsp\_rate**/2.0 Hz. It is acceptable to specify the same frequency multiple times; however, if different gains are specified for the same frequency, it is indeterminate as to which of the gains will be applied at that frequency. Frequency/Gain pairs do not necessarily have to be specified in order of increasing frequency. If they are not, they will be sorted into the appropriate order prior to application.

**G**: pointer to a corresponding array of gain values (dB). The 0<sup>th</sup> element is used to specify a "preamp gain" value to be applied to all frequencies. The G[] array has (**nfreqs** + 1) values.

For convenience and consistency with legacy consoles, two "short-cut" calls with pre-specified frequencies are provided.

**void SetRXAGrphEQ (int channel, int\* rxeq)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**rxeq**: pointer to an array containing gain values (dB) corresponding to an implicit "F vector" of F[] = {0.0, 150.0, 400.0, 1500.0, 6000.0}. The value in G[0] is the "preamp gain" to be applied to all frequencies. G[1] through G[4] contain gains corresponding to 150.0 Hz through 6000.0 Hz, respectively.

**void SetRXAGrphEQ10 (int channel, int\* rxeq)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

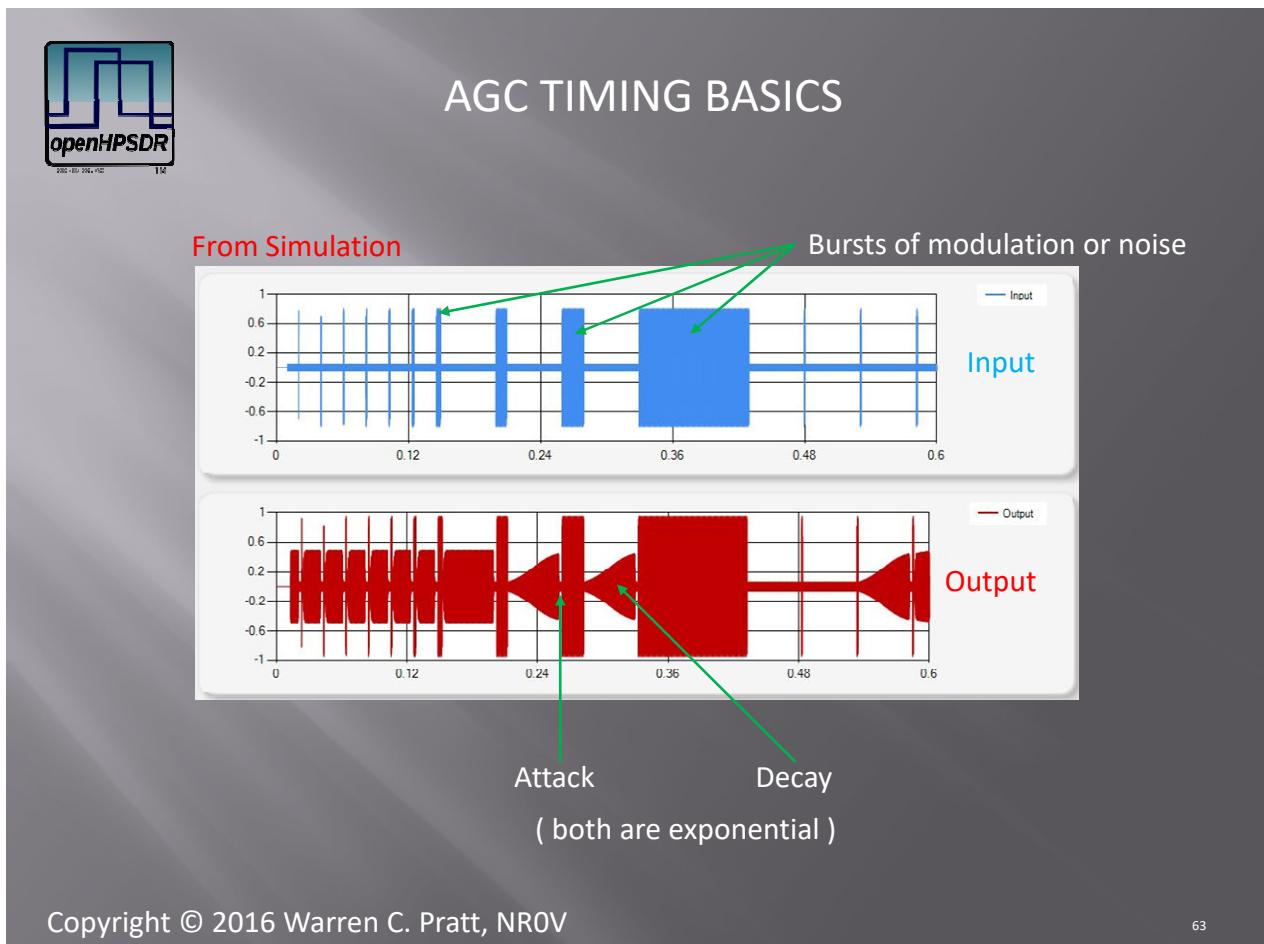
**rxeq**: pointer to an array containing gain values (dB) corresponding to an implicit "F vector" of F[] = {0.0, 32.0, 63.0, 125.0, 250.0, 500.0, 1000.0, 2000.0, 4000.0, 8000.0, 16000.0}. The value in G[0] is the "preamp gain" to be applied to all frequencies. G[1] through G[10] contain gains corresponding to 32.0 Hz through 16000.0 Hz, respectively.

## AGC

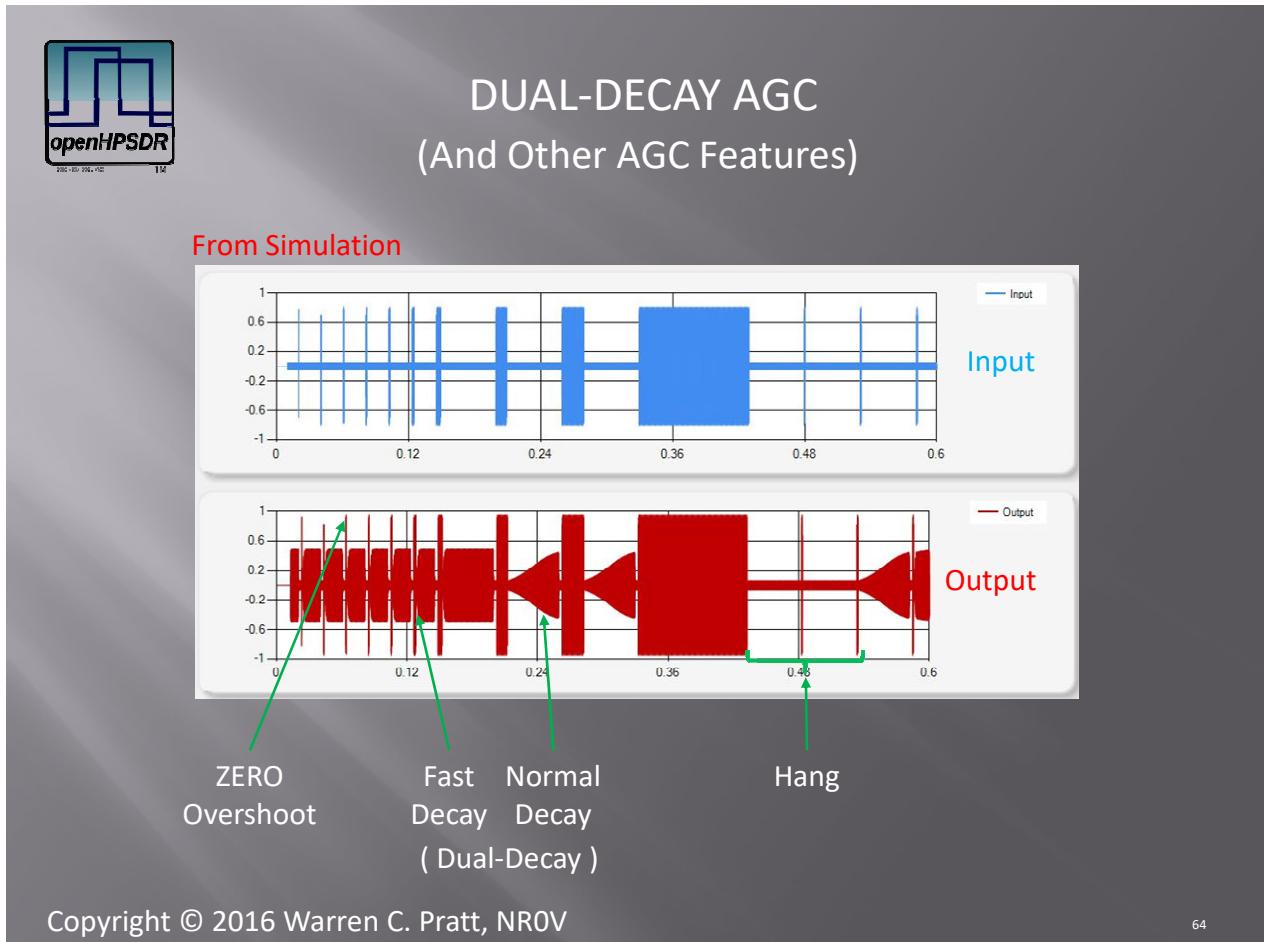
WDSP provides a full-featured AGC including:

- Exponential attack / decay
- Hang for quieting when listening to relatively strong stations
- Dual-decay for suppression of certain impulses not removable using wideband noise blankers
- Controls for on-panadapter adjustment of the AGC Threshold and Hang Level
- Zero Overshoot for avoidance of un-natural sound and AGC "pumping"
- Slope to allow stronger signals to yield higher audio output than weaker signals

The following figure illustrates the exponential attack and decay. The attack is typically set to be much faster than the normal decay. Note that the blue upper plot is of the AGC input signal and the lower red plot is of the AGC output signal.

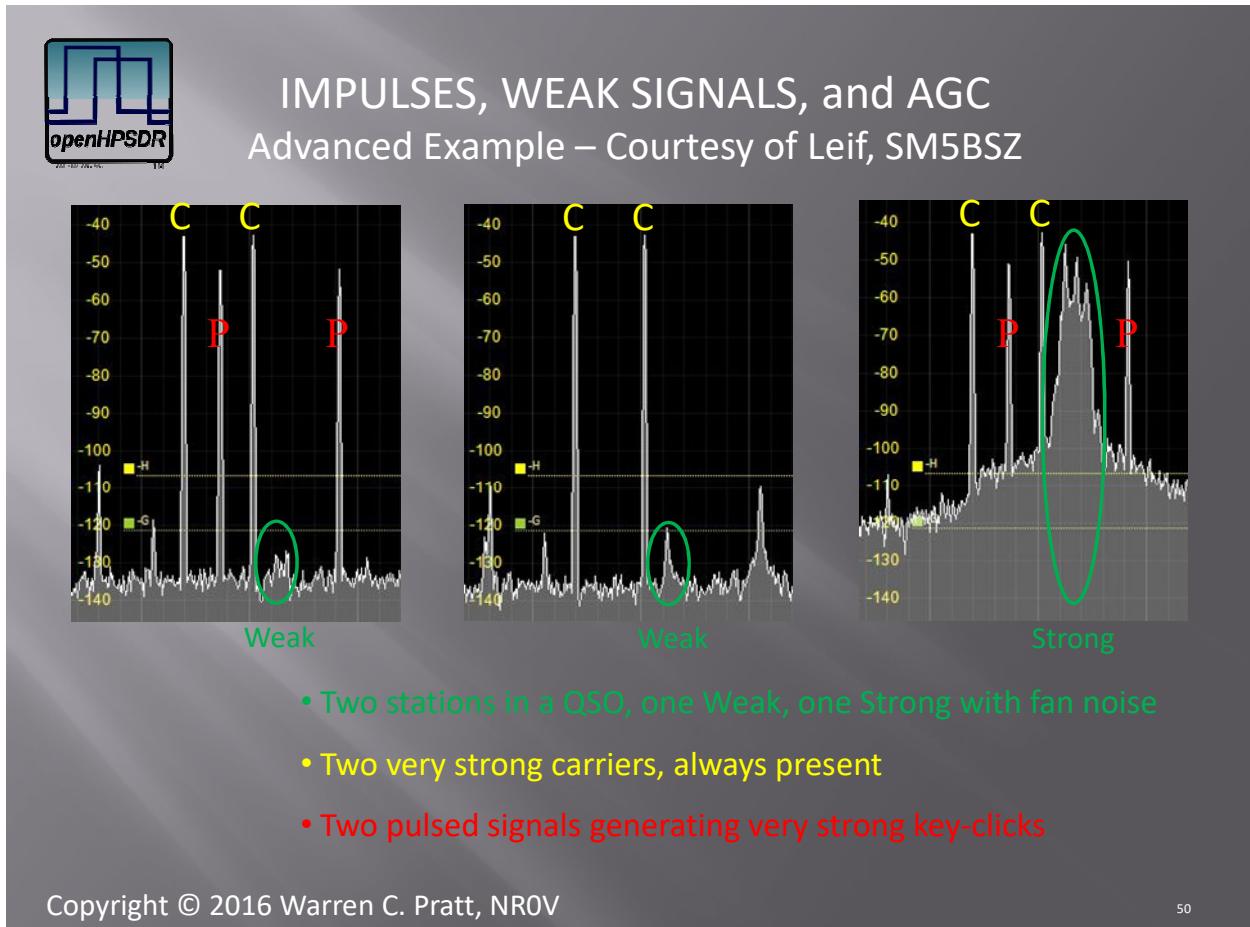


The following figure illustrates additional features of Zero Overshoot, Dual-Decay, and Hang. The Dual-Decay is used in the suppression of certain impulses that cannot be removed with wideband noise blankers. If the AGC judges a short burst to be an impulse, a fast-decay is invoked to avoid the AGC from blocking the reception of weak signals for an extended time following the impulse. Otherwise, the normal decay is used. Hang maintains the gain at a constant fixed level for a specified amount of time after a strong burst of reception.



The following figure demonstrates a situation in which the Dual-Decay feature provides a major improvement in reception. The IQ.wav file for this example is available from Leif Asbrink, SM5BSZ, and is posted on his website.

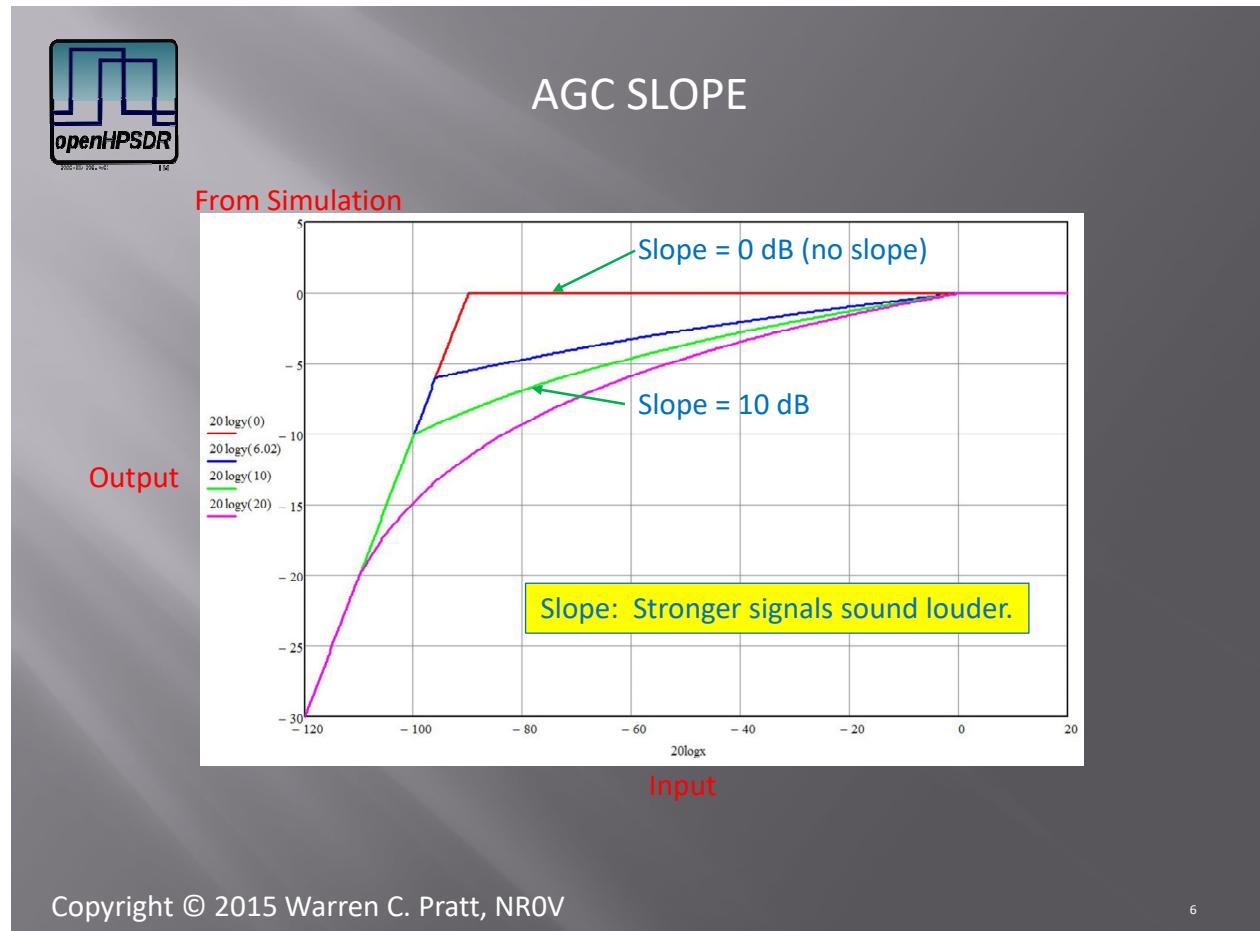
Shown are snapshots of the frequency spectrum taken at three different times. The two strong carriers render wideband noise blankers useless. The key clicks from the two pulsed carriers are the impulses we want to suppress. Strong fan noise on the strong signal implies that we need a relatively long decay as the normal decay for the AGC such that the fan noise is suppressed between words and phrases of the strong station. However, a long AGC time-constant would cause the AGC to block the weak station for an extended time after each impulse. Fortunately, the fast decay is invoked after the impulses and the weak station is not blocked.



The following figure illustrates the AGC Gain curve with four different values of slope: 0 dB (no slope), 6 dB, 10 dB, and 20 dB. The plot shows AGC Output level versus AGC Input level, both in dBm. The AGC maximum gain was set to 90 dB.

With no slope, all input levels above -90 dBm have the same output level, the AGC adjusts its gain to produce the same output. Since the maximum gain is set to 90 dB, the output drops (-10 dBm out for -10 dBm in) for input values lower than -90 dBm.

At a slope of 10dB, the output level gradually increases from -10 dBm to 0 dBm for input levels of -100 dBm to 0 dBm.



The following call is used to both (1) turn AGC OFF/ON, and (2) in the event it's ON, set some key parameters:

```
void SetRXAAGCMode (int channel, int mode)
```

**channel**: number of the receiver channel being accessed

**mode**: 0: AGC\_OFF. A fixed gain equal to the AGC fixed gain setting will be applied to the signal.

1: AGC\_LONG. Hangtime = 2000 ms. Decay\_time\_constant = 2000 ms.  
2: AGC\_SLOW. Hangtime = 1000 ms. Decay\_time\_constant = 500 ms.  
3: AGC\_MED. Hang is turned OFF. Decay\_time\_constant = 250 ms.  
4: AGC\_FAST. Hang is turned OFF. Decay\_time\_constant = 50 ms.  
5: AGC\_CUSTOM. AGC is ON; however, no parameters are set by this call. All parameters are set with other calls as needed.  
[default = 3]

The following three calls can be used for setting various AGC time-constants or absolute times.

#### **void SetRXAAGCAttack (int channel, int attack)**

**channel**: number of the receiver channel being accessed

**attack**: attack time constant (ms). It is recommended to leave this at the default value. Note that "attacks" are along exponential curves and this value is a time-constant as opposed to an absolute amount of time.

[default = 1 ms]

#### **void SetRXAAGCDecay (int channel, int decay)**

**channel**: number of the receiver channel being accessed

**decay**: decay time constant (ms). Note that "decays" are along exponential curves and this value is a time-constant as opposed to an absolute amount of time.

[default = 250 ms]

#### **void SetRXAAGCHang (int channel, int hang)**

**channel**: number of the receiver channel being accessed

**hang**: hang-time (ms). Note that this is an absolute amount of time as opposed to a time-constant.

[default = 250 ms]

Support is provided for setting the AGC Hang Level either on the bandscope or by use of a simple slider control. When one is changed, the other needs to be adjusted accordingly. Therefore, there are both 'get' and 'set' commands for each type of control. For example, when a 'set' is done on the slider, there is a 'get' that can be used to return the new position required on the bandscope.

#### **void GetRXAAGCHangLevel (int channel, double\* hanglevel)**

**channel**: number of the receiver channel being accessed

**hanglevel**: pointer to location where the bandscope level (dBm) is to be stored.

#### **void SetRXAAGCHangLevel (int channel, double hanglevel)**

**channel**: number of the receiver channel being accessed

**hanglevel**: bandscope level being set as the hang-level (dBm)

**void GetRXAAGCHangThreshold (int channel, int\* hangthreshold)**

**channel**: number of the receiver channel being accessed

**hangthreshold**: pointer to location where the slider setting is to be stored (setting range is 0 to 100)

**void SetRXAAGCHangThreshold (int channel, int hangthreshold)**

**channel**: number of the receiver channel being accessed

**hangthreshold**: slider setting being set as the hang-level (setting range is 0 to 100)

Support is provided for setting the AGC Maximum-Gain / Threshold both via a slider and by moving a line on the bandscope. When one is changed, the other needs to be adjusted accordingly. Therefore, there are both 'get' and 'set' commands for each type of control. For example, when a 'set' is done on the slider, there is a 'get' that can be used to return the new position required on the bandscope.

On the bandscope, the noise floor moves up and down depending upon the FFT\_size and sample\_rate of data used to compute the bandscope. Also, it is useful to reference the bandscope line to the noise floor, typically setting the line a few dB above the noise floor. Therefore, the display FFT\_size and display sample\_rate parameters must be supplied to achieve the correct bandscope results.

**void GetRXAAGCThresh (int channel, double\* thresh, double size, double rate)**

**channel**: number of the receiver channel being accessed

**thresh**: pointer to location to store the bandscope level (dBm) for AGC Threshold

**size**: FFT\_size being used to compute the bandscope display

**rate**: sample-rate of data being supplied to compute the bandscope display

**void SetRXAAGCThresh (int channel, double thresh, double size, double rate)**

**channel**: number of the receiver channel being accessed

**thresh**: the bandscope level (dBm) for AGC Threshold

**size**: FFT\_size being used to compute the bandscope display

**rate**: sample-rate of data being supplied to compute the bandscope display

**void GetRXAAGCTop (int channel, double\* max\_agc)**

**channel**: number of the receiver channel being accessed

**max\_agc**: pointer to location to store the maximum AGC Gain, i.e., the AGC Threshold (range is -20 dB to +120dB)

**void SetRXAAGCTop (int channel, double max\_agc)**

**channel**: number of the receiver channel being accessed

**max\_agc**: the maximum AGC Gain, i.e., the AGC Threshold (range is -20 dB to +120dB)  
[default = +80 dB]

**void SetRXAAGCSlope (int channel, int slope)**

**channel**: number of the receiver channel being accessed

**slope**: Ten times the desired AGC slope value (dB). The typical range for slope would be 0 to 200, corresponding to 0dB to 20dB.  
[default = 35, corresponding to 3.5dB]

**void SetRXAAGCFixed (int channel, double fixed\_agc)**

**channel**: number of the receiver channel being accessed

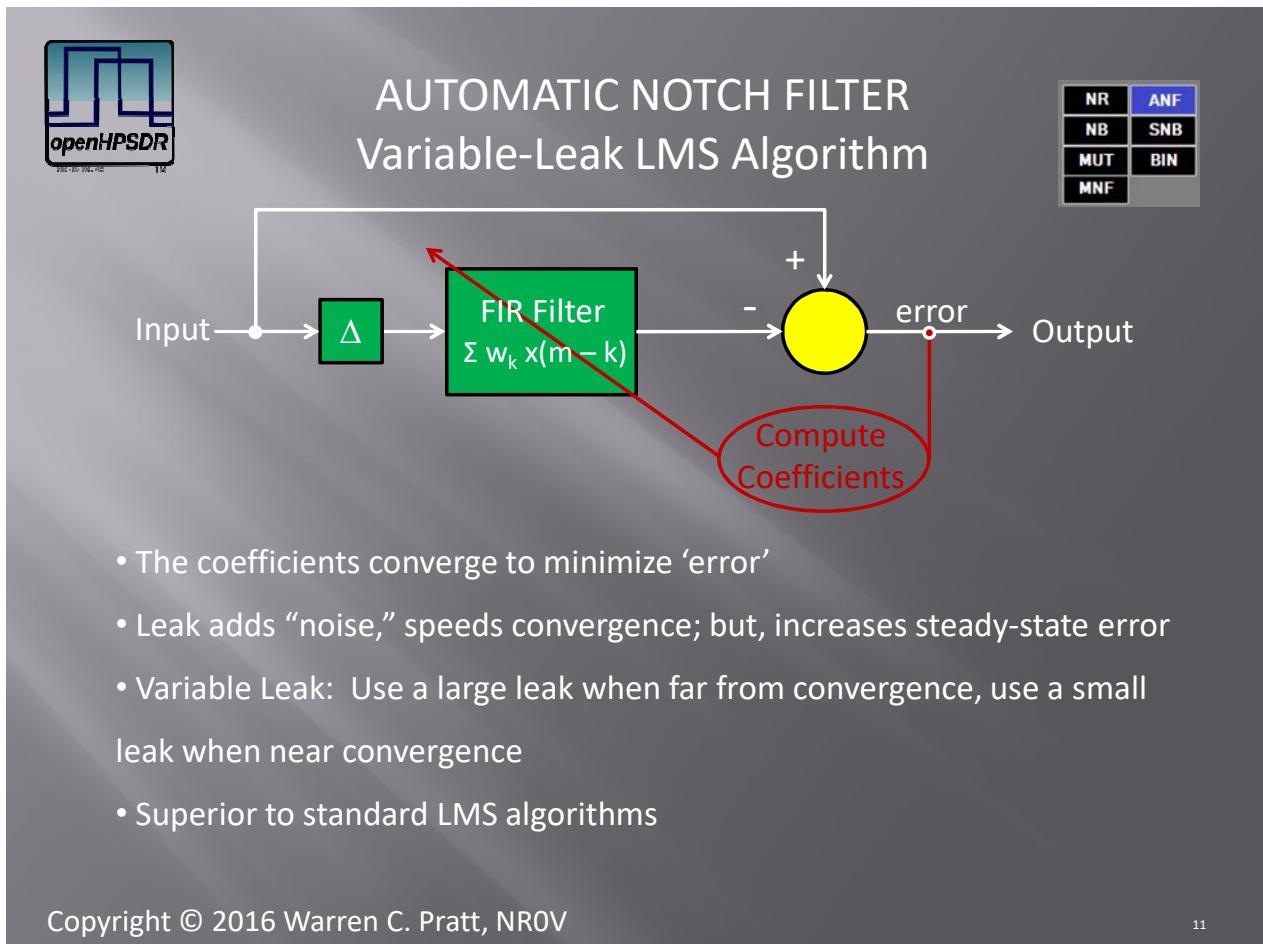
**fixed\_agc**: fixed gain to use when the AGC is OFF, i.e., the '**mode**' is set to 0. (dB)  
[default = +60 dB]

## Automatic Notch Filter

The WDSP Automatic Notch Filter is used primarily to "notch-out" carriers and similar signals.

The Least-Mean-Squares (LMS) algorithm used here dynamically computes the coefficients of an FIR filter such that the filter passes the carrier and rejects other signals. The carrier is then subtracted from the original signal to produce an output free of the carrier.

The filter coefficients are computed in such a way as to maximize the autocorrelation of the carrier. The delay,  $\Delta$ , assists in the comparison of two different time periods of the signal.



The Automatic Notch Filter is turned OFF/ON using the call:

```
void SetRXAANFRun (int channel, int run)
```

**channel:** number of the receiver channel being accessed

**run:** 0 = OFF; 1 = ON  
[default = 0]

The number of FIR filter taps is set with the call:

**void SetRXAANFTaps (int channel, int taps)**

**channel**: number of the receiver channel being accessed

**taps**: number of filter taps (coefficients) to be used  
[default = 64]

The delay is set using the call:

**void SetRXAANFDelay (int channel, int delay)**

**channel**: number of the receiver channel being accessed

**delay**: delay, specified in number of samples  
[default = 16]

**void SetRXAANFGain (int channel, double gain)**

**channel**: number of the receiver channel being accessed

**gain**: gain value. Typical range is 1.0e-06 to 1.0e-03.  
[default = 1.0e-04]

Note that increasing gain allows the algorithm to more effectively lock-onto and notch carriers. HOWEVER, at higher gains the algorithm will begin to generate distortion on speech. I.e., there is a tradeoff between notching effectiveness and speech distortion.

**void SetRXAANFLeakage (int channel, double leakage)**

**channel**: number of the receiver channel being accessed

**leakage**: leak value. Typical range is 0.001 to 1.000.  
[default = 0.100]

The following call can be used to set **taps**, **delay**, **gain**, and **leakage** using only a single call:

**void SetRXAANFVals (int channel, int taps, int delay, double gain, double leakage)**

**channel**: number of the receiver channel being accessed

**taps**: number of filter taps (coefficients) to be used  
[default = 64]

**delay**: delay, specified in number of samples  
[default = 16]

**gain**: gain value. Typical range is 1.0e-06 to 1.0e-03.  
[default = 1.0e-04]

**leakage**: leak value. Typical range is 0.001 to 1.000.  
[default = 0.100]

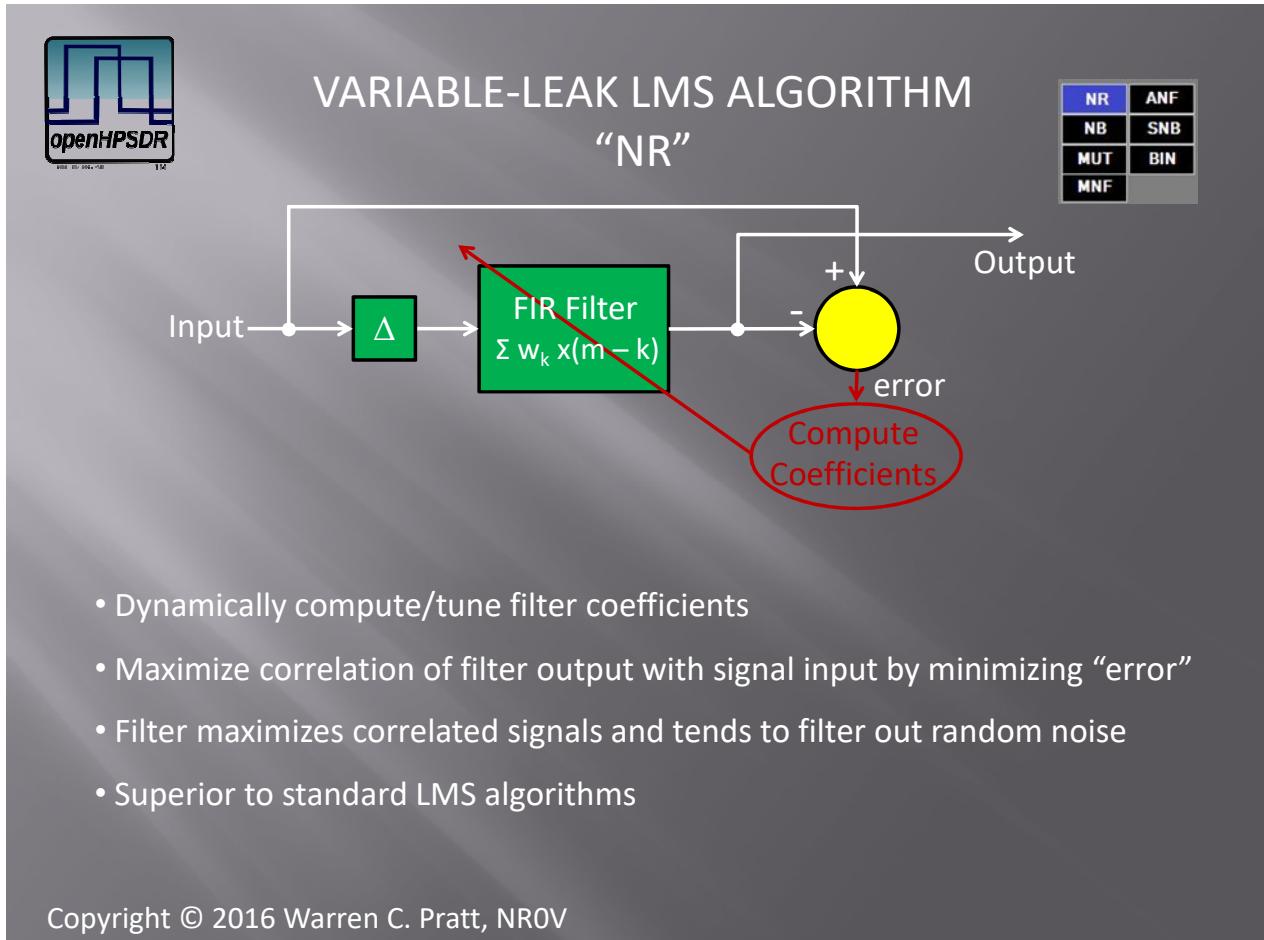
**void SetRXAANFPosition (int channel, int position)**

**channel**: number of the receiver channel being accessed

**position**: 0 = execute ANF before AGC; 1 = execute ANF after AGC.

## LMS Noise Reduction

Setting your expectations: LMS Noise Reduction algorithms work best when there is already a good signal-to-noise ratio; they can be very effective in providing a quiet background under these circumstances. This is really their purpose. It is unlikely that you will be able to pull an otherwise unreadable signal out of the noise with this type algorithm.



The LMS Noise Reduction is turned OFF/ON using the call:

```
void SetRXAANRRun (int channel, int run)
```

**channel**: number of the receiver channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 0]

The number of FIR filter taps is set with the call:

```
void SetRXAANRTaps (int channel, int taps)
```

**channel**: number of the receiver channel being accessed

**taps**: number of filter taps (coefficients) to be used  
[default = 64]

The delay is set using the call:

**void SetRXAANRDelay (int channel, int delay)**

**channel**: number of the receiver channel being accessed

**delay**: delay, specified in number of samples  
[default = 16]

**void SetRXAANRGain (int channel, double gain)**

**channel**: number of the receiver channel being accessed

**gain**: gain value. Typical range is 1.0e-06 to 1.0e-03.  
[default = 1.0e-04]

**void SetRXAANRLeakage (int channel, double leakage)**

**channel**: number of the receiver channel being accessed

**leakage**: leak value. Typical range is 0.001 to 1.000.  
[default = 0.100]

The following call can be used to set **taps**, **delay**, **gain**, and **leakage** using only a single call:

**void SetRXAANRVals (int channel, int taps, int delay, double gain, double leakage)**

**channel**: number of the receiver channel being accessed

**taps**: number of filter taps (coefficients) to be used  
[default = 64]

**delay**: delay, specified in number of samples  
[default = 16]

**gain**: gain value. Typical range is 1.0e-06 to 1.0e-03.  
[default = 1.0e-04]

**leakage**: leak value. Typical range is 0.001 to 1.000.  
[default = 0.100]

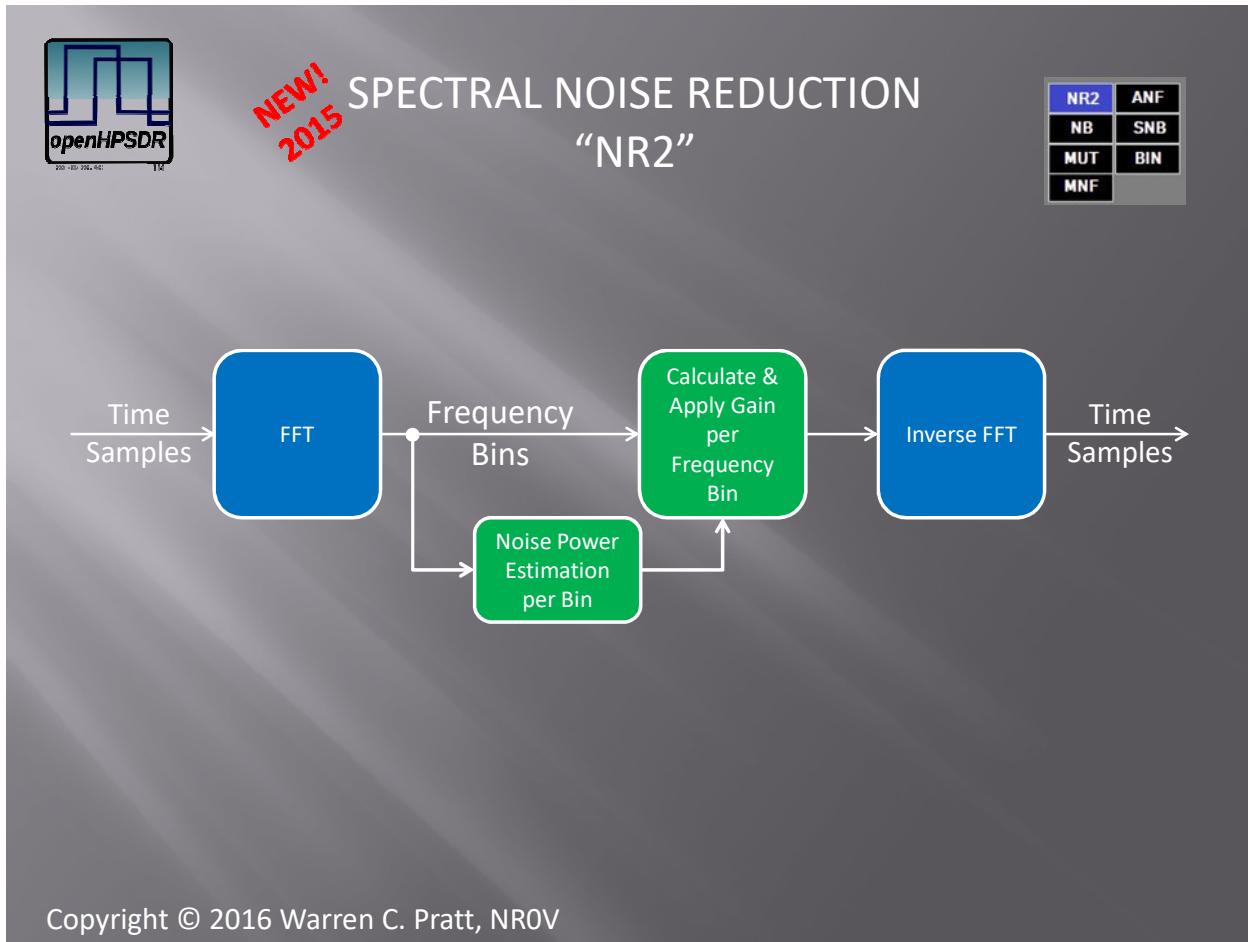
**void SetRXAANFPosition (int channel, int position)**

**channel**: number of the receiver channel being accessed

**position**: 0 = execute Noise Reduction before AGC; 1 = execute Noise Reduction after AGC.

## Spectral Noise Reduction

The seminal work for the class of algorithm implemented here for the Spectral Noise Reduction block was published in 1984<sup>1</sup>. At a basic level, an FFT is done to convert to the frequency domain, a gain is applied to each of the frequency bins based upon estimates of the amplitudes of speech and noise within the bins, and an inverse FFT is done to get back to time-domain samples. Statistical models of speech and noise amplitudes are used as "a priori" knowledge to enhance the accuracy of the estimates.



Over the years, various alternatives and improvements to the speech estimation algorithms and corresponding gain calculations have been developed. Four variants are included in this code and are selectable. Also, noise estimation is an important part of the process and three different noise estimation algorithms have been included.

### A Word About Using Noise Reduction

For optimal operation, noise reduction algorithms should be provided with the best possible environment within the radio's processing chain. If using Post-AGC NR, the AGC Threshold should be as far as practical above the noise floor. A Pre-AGC position will generally offer the best signal-to-noise

<sup>1</sup> Yariv Ephraim and David Malah, "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator," 1984.

ratio going into the NR, which is beneficial; however, a Post-AGC position may offer better immunity to sudden noise changes. These factors should be taken into consideration anytime any of these noise reduction algorithms are in use.

The Spectral Noise Reduction is turned OFF/ON using the call:

**void SetRXAEMNRRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

**run**: 0 = OFF; 1 = ON

[default = 0]

The gain-per-bin calculation method is selected using the following call. The first three choices will produce somewhat similar results with the default Gamma speech distribution being somewhat preferred.

The fourth method, a Trained algorithm, is distinct and can produce more dramatic noise reduction; albeit, at the price of producing the human perception of being harsh – more on post-processing for this later. This Trained algorithm was trained to accurately determine, by FFT bin, the signal-to-noise ratio by evaluating multiple “views” of the incoming signal and noise. About seventy-two hours of signal and noise were used for training.

**void SetRXAEMNRgainMethod (int channel, int method)**

**channel**: number of the receiver channel being accessed

**method**: 0 = Gaussian speech distribution, linear amplitude scale; 1 = Gaussian speech distribution, log amplitude scale; 2 = Gamma speech distribution; 3 = Trained algorithm

[default = 2]

The noise-power-estimation (per bin) method is selected using the following call. All three choices will produce somewhat similar results. The OSMS method has been long-proven and is preferred by many. The MMSE method is more responsive to changing noise than OSMS. The NSTAT method is the most responsive of all and may be preferred with rapidly changing noise – it is especially recommended with the “Trained” gain method.

**void SetRXAEMNRnpeMethod (int channel, int method)**

**channel**: number of the receiver channel being accessed

**method**: 0 = Optimal Smoothing Minimum Statistics (OSMS); 1 = Minimum Mean-Square Error (MMSE); 2 = Non-stationary Noise Method (NSTAT)

[default = 0]

The position in the processing pipeline where this algorithm is applied is selected by:

### **void SetRXAEMNRPosition (int channel, int position)**

**channel:** number of the receiver channel being accessed

**position:** 0 = execute noise reduction before AGC; 1 = execute noise reduction after AGC  
[default = 0]

*Specifically for the “Trained” mode,* there are Threshold adjustments that determines at what level and how the training data is invoked. Said differently, the thresholds determine at what SNR the signal will be allowed to pass. The higher the “T1” threshold, the better the signal-to-noise ratio must be for the signal to pass through. Therefore, if you want weaker signals and perhaps a little noise, choose a lower value.

### **void SetRXAEMNRtrainZetaThresh (int channel, double thresh)**

**channel:** number of the receiver channel being accessed

**thresh:** Suggested range is -5.0 to +5.0.  
[default = -0.5]

In its quest to eliminate all noise, the “Trained” mode can sometimes also eliminate very weak signals that it does not separate from the noise. By lowering the value of an additional threshold, called “T2”, those signals may be allowed to pass at the cost of also allowing some additional noise.

### **void SetRXAEMNRtrainT2 (int channel, double t2)**

**channel:** number of the receiver channel being accessed

**thresh:** Suggested range is 0.02 to 0.30.  
[default = 0.20]

## POST-PROCESSING

Following the basic algorithms, as described above, there are a number of post-processing steps that may be applied to improve the human perception of the processed speech.

This first technique, an Artifact-Elimination Post-Filter (AEPF), should ALWAYS be applied with these types of algorithms.

Noise reduction algorithms that operate in the frequency domain and that individually modify FFT frequency bins are well-known for producing artifacts often called “musical tones.” This filter is a dynamically-self-adjusting filter to largely remove those tones. There is no reason to turn it off, other than for development test purposes.

### **void SetRXAEMNRaeRun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 = OFF; 1 = ON  
[default =1]

The psychoacoustics of audio processed by various NR algorithms can also often be improved by injecting various types of noise and weak-signals and then tapering the frequency response of the signal. Several capabilities are provided to do this.

This following call turns ON/OFF the following set of noise-related post-processing functions. The default is set to zero which directs this set of functions to do absolutely nothing in the event the console controls for them are not implemented.

#### **void SetRXAEMNRpost2Run (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 = OFF; 1 = ON  
[default =0]

For psychoacoustic reasons, noise (of a couple types) and signals can be injected into the audio stream coming from the fundamental noise reduction algorithms discussed above. This can have the effect of allowing very weak signals through and it creates the perception of “softening” the audio sometimes providing more natural-sounding speech.

The level of the noise/signals to be mixed into the audio stream is adjusted by a control:

#### **void SetRXAEMNRpost2Nlevel (int channel, double nlevel)**

**channel:** number of the receiver channel being accessed

**nlevel:** range is 0.0 (minimum injection) to 100.0 (maximum injection)  
[default = 15]

The composition of the noise/signals can also be varied, depending upon personal preference of the operator. At the two extremes are (1) a sample of the noise/signal before the noise reduction algorithm is applied, and (2) white noise with some “speech shaping”. Any mix between the two extremes is possible, i.e., all from the original signal OR all white OR any mix in between. The amplitude of the white noise is also automatically adjusted based upon the strength of the received signal.

#### **void SetRXAEMNRpost2Factor (int channel, double factor)**

**channel:** number of the receiver channel being accessed

**factor:** range is 0.0 (sample of original signal/noise) to 100.0 (white noise)  
[default = 15]

The white noise, if injected, has no signal-value; although, as mentioned, it may serve to “soften” the audio and enhance the human listening experience. That said, it is of no value at all if there is no signal

present and may become tiring to the listener. Hence, there is a means to allow it to be present only during speech or to fade-away over the longer-term in the absence of speech.

This control expresses a time-constant in seconds and determines the fade-rate for specifically any white noise that has been injected. At a setting of 0.0, the noise will basically ONLY exist during the syllables of the speech to soften the audio. At a setting of 100.0, specifically the white noise will decline to 37% (one time-constant) of its original value after 100 seconds. Hence, during speech, it remains present.

#### **void SetRXAEMNRpost2Rate (int channel, double tc)**

**channel**: number of the receiver channel being accessed

**tc**: range is 0.0 (fastest noise decline) to >=100.0 (slowest noise decline)  
[default = 5.0, operator may like it or find it very annoying]

Finally, to remove residual artifacts and soften the effects of noise, a “tapering” of the frequency response of the audio has been found to be desirable. This is not a “bandpass” in the usual sense in that the frequency response is “rolled-off” along a specific response curve. The upper-frequency cut-off of this curve can be adjusted.

#### **void SetRXAEMNRpost2Taper (int channel, int taper)**

**channel**: number of the receiver channel being accessed

**taper**: range is 0 (EVERYTHING is cut off) to 100 (maximum audio bandwidth, normally 24 KHz)  
[default =12, good for normal SSB, use higher for ESSB or wideband AM]

#### **ADDITIONAL OPTIONS & REQUIREMENTS:**

- At initialization, the Spectral Noise Reduction algorithm looks for the file "calculus" located in the same directory as the WDSP executable. "calculus" is a large data file containing information required for the best (also the default) gain-computation algorithm choice. This data can be read from a file to enable future experimentation with different and/or improved data.

If the file is not found, a substitute set of data that is stored in the executable will be used. This data is the same as that of the file as of the writing of version 1.27 of this document.

- Specifically for the “Trained” gain mode, the algorithm looks for the file “zetaHat.bin”. This file contains the essential data from the training of the algorithm. This file would normally be copied into the same directory as the WDSP executable.

If the file is not found, a substitute set of data that is stored in the executable will be used. This data is the same as that of the file as of the writing of version 1.27 of this document.

## Bandpass Filter

For certain modes and features, a second bandpass filter is needed. It is automatically activated when required. Response curves are the same as the Notched Bandpass Filter discussed above.

It's lower and upper frequencies can be set using the following **DEPRECATED** call. However, the preferred method is by calling the RXA Collective **RXASetPassband(...)**.

**void SetRXABandpassFreqs (int channel, double f\_low, double f\_high)** **DEPRECATED**

**channel**: number of the receiver channel being accessed

**f\_low**: lower of the two frequencies (Hertz). May be negative or positive.  
[default = -4150.0]

**f\_high**: higher of the two frequencies (Hertz). May be negative or positive.  
[default = -150.0]

The window function used in designing the filter can be selected with the following call. See the discussion (preceding) on the Notched Bandpass Filter for more information on this selection.

**void SetRXABandpassWindow (int channel, int wintype)**

**channel**: number of the receiver channel being accessed

**wintype**: 0 for 4-term Blackman-Harris window; 1 for 7-term Blackman-Harris window.  
[default = 1]

The length of the desired filter impulse response *can* be set with the following call. This is NOT the preferred method. Instead, a "Collective" call can be used. (See the section on "RXA Collectives.") See the discussion (preceding) on the Notched Bandpass Filter for more information on this selection.

**void SetRXABandpassNC (int channel, int nc)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**nc**: length of the desired filter impulse response. This must be a power of two AND must be greater than or equal to the DSP Buffer Size in use.  
[default = 2048]

The Filter-Type (Linear Phase or Low Latency) for the Bandpass Filter can be set using the following call. This is NOT the preferred method. Instead, a "Collective" call can be used. See the section on "RXA Collectives."

**void SetRXABandpassMP (int channel, int mp)**

**DEPRECATED**

**channel**: number of the receiver channel being accessed

**mp**: 0 for Linear Phase; 1 for Low Latency (Minimum Phase)

[default = 0]

## Scope/Phase Display Sender

The console can call for buffers of processed samples to use for things like an oscilloscope or phase display. Two calls are provided for this purpose. One returns only the 'I' samples while the other returns interleaved I/Q samples.

**void RXAGetaSipF (int channel, float\* out, int size)**

**channel:** number of the receiver channel being accessed

**out:** pointer to buffer into which the I-only samples (float format) are to be placed

**size:** number of I-only samples to retrieve. '**size**' must be <= 4096 per an internal setting.

**void RXAGetaSipF1 (int channel, float\* out, int size)**

**channel:** number of the receiver channel being accessed

**out:** pointer to buffer into which the interleaved I/Q samples (float format) are to be placed

**size:** number of I/Q samples to retrieve. '**size**' must be <= 4096 per an internal setting.

## AM Carrier Block

The AM carrier, a DC value, is left in the signal through the AGC. This is desirable such that the typical "AGC-quieting" of an AM receiver will be preserved. However, there are situations where it is desirable to remove this carrier before the WDSP audio output. One example would be when an AM signal is being recorded with the intention of playing it back through the transmitter. In that case, if the carrier were left in the signal, the carrier would be recorded also and the carrier level would be too high on playback through the transmitter since the AM transmitter would add carrier again.

This is a simple filter which removes this carrier, i.e., DC component.

The AM Carrier Block is turned OFF/ON using the call:

**void SetRXACBLRun (int channel, int run)**

**channel**: number of the receiver channel being accessed

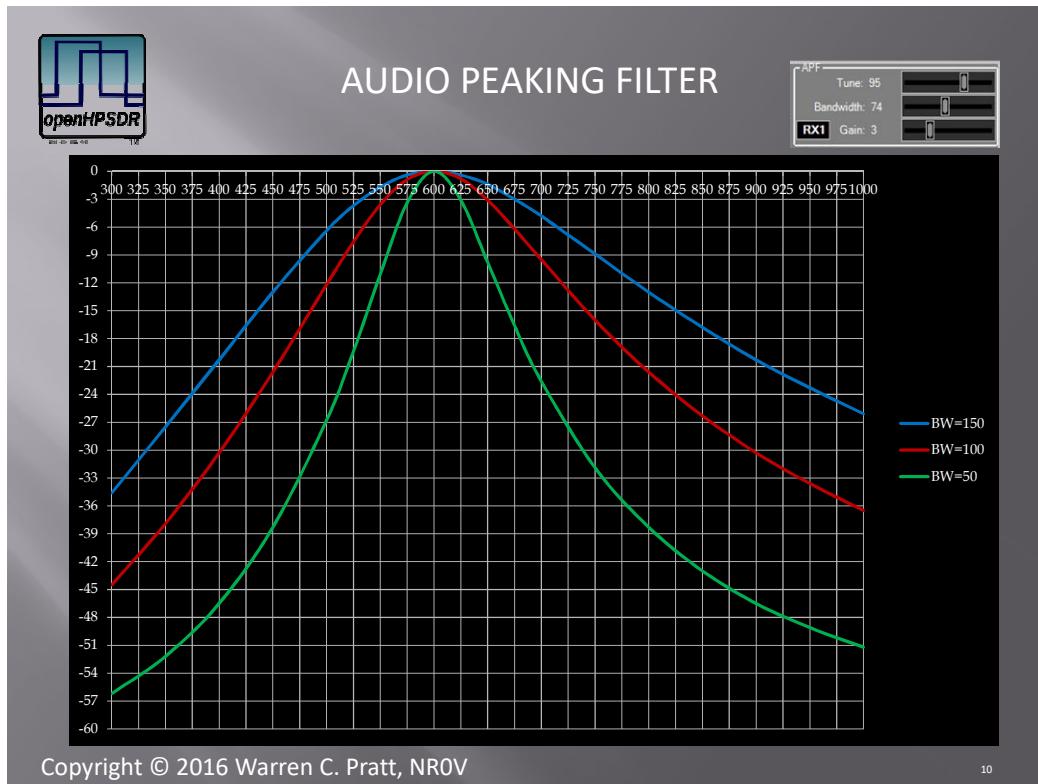
**run**: 0 = OFF; 1 = ON

[default = 0]

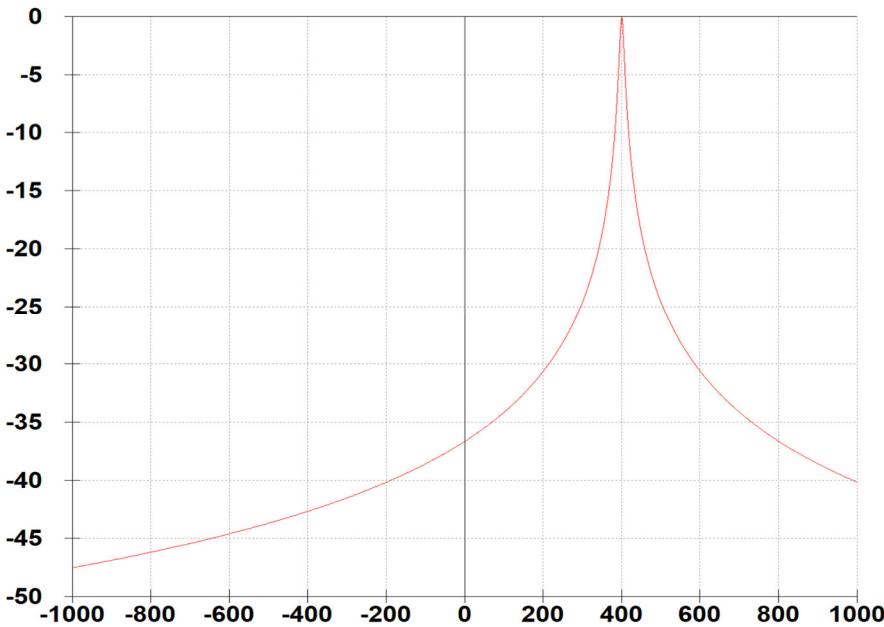
## CW Peaking Filter

The CW Peaking Filter is used to peak a single frequency, the frequency of a desired CW signal. Gain, bandwidth, and tune frequency are all selectable. There are four different filter types available:

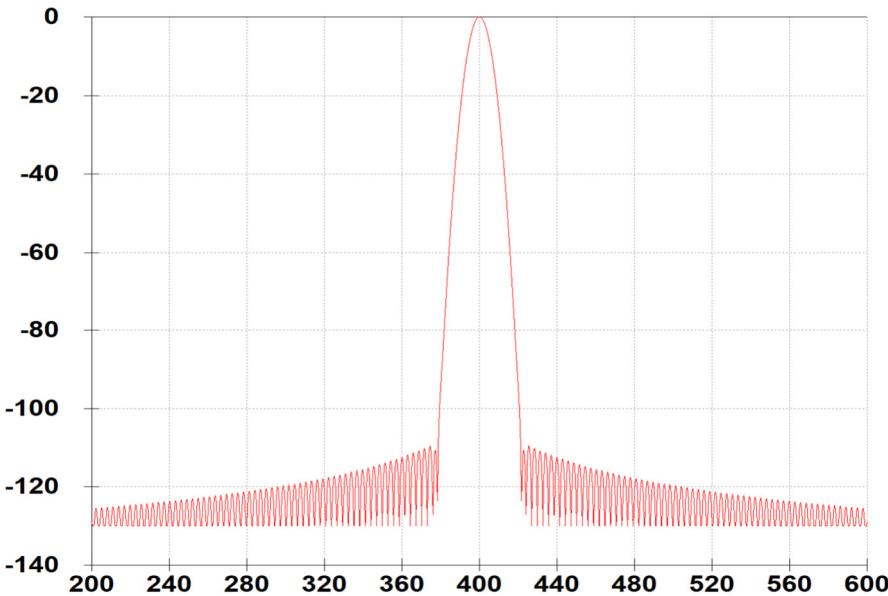
- (1) Bi-Quad: A four-stage bi-quad IIR filter. This was the original APF filter in WDSP and has low-latency. Examples (from measurements) of the frequency response are shown in the figure below.



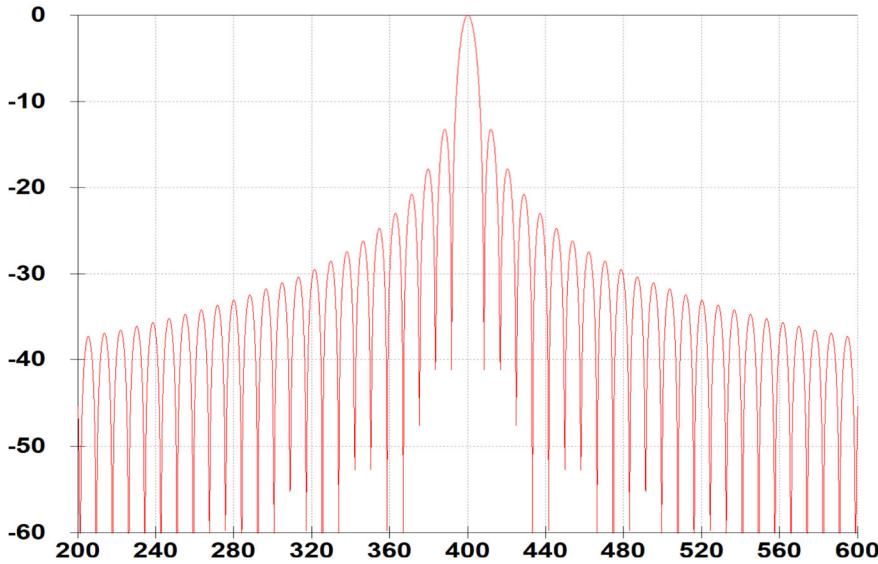
- (2) Double-Pole: This filter is modelled from a two-pole low-pass filter with very high Q. The frequency response of such a filter has a sharp peak at its cut-off frequency. The CW signal is tuned to be within this peak. Listening tests have shown it to be very desirable in terms of response and listening comfort. Shown below is the response of a Double-Pole filter tuned to 400Hz and set for a 20Hz -6dB bandwidth.



- (3) Gaussian: Gaussian filters have the unique property that they have zero ringing. They also have a desirable shape-factor for selection of a single CW signal. Therefore, this option is included. The frequency response is, of course, Gaussian in shape. Shown here is the response of a Gaussian filter tuned to 400Hz and with a -6dB bandwidth of 10Hz.



- (4) Matched Filter: The matched filter is a very special type of filter often used for weak-signal communications. It has the potential to provide the best signal-to-noise ratio. However, tuning is critical, i.e., the tuned frequency of the filter must match the pitch of the received tone. Its frequency response appears rather unusual because, in the time-domain, it essentially represents the convolution of the CW signal with itself. The frequency response of a Matched Filter with a center frequency of 400Hz and a -6dB bandwidth of 10Hz is shown below.



About using these filters:

- Many operators will find it desirable to set the bandwidth to something in the range of 50Hz. Setting the filters to very narrow bandwidths may prevent hearing/detecting nearby signals.
- For situation where the CW signal is nearly buried in the band-noise, it may be desirable to reduce the bandwidth to 20Hz or even below. However, with such narrow settings, there is not adequate bandwidth to receive fast CW with the CW symbols being “smeared” together. Therefore, this is best for slow speeds, e.g., 12 WPM.

The CW Peaking Filter is turned OFF/ON with the call:

**void SetRXASPCWRun (int channel, int run)**

**channel:** number of the receiver channel being accessed

**run:** 0 = OFF; 1 = ON  
[default = 0]

Filter Selection is set with the following call:

**void SetRXASPCWSelection (int channel, int selection)**

**channel:** number of the receiver channel being accessed

**selection:** 0 = Double-Pole; 1 = Matched; 2 = Gaussian; 3 = Bi-Quad

Center frequency, bandwidth, and gain are set using the following three calls, respectively.

**void SetRXASPCWFreq (int channel, double freq)**

**channel:** number of the receiver channel being accessed

**freq**: center frequency in Hertz  
[default = 600.0]

**void SetRXASPCWBandwidth (int channel, double bw)**

**channel**: number of the receiver channel being accessed  
**bw**: approximate -6dB bandwidth in Hertz  
[default = 100.0]

**void SetRXASPCWGain (int channel, double gain)**

**channel**: number of the receiver channel being accessed  
**gain**: this is an absolute gain factor (multiplier) -- NOT dB.  
[default = 2.0]

## Dolly Filter

The Dolly Filter is a dual-lobe filter intended for use in RTTY reception. It comprises a pair of CW peaking filters whose inputs are the same and whose outputs are summed. Since it is set up for RTTY operation, only two peaks are desired; however, the underlying filter structure supports an arbitrary number of peaks, each with its own Enable, Center\_Frequency, Bandwidth, and Gain.

**void SetRXAmpeakRun (int channel, int run)**

channel: number of the receiver channel being accessed

run: 0 = OFF; 1 = ON

[default = 0]

**void SetRXAmpeakNpeaks (int channel, int npeaks)**

channel: number of the receiver channel being accessed

npeaks: number of peaks (peaking filters) desired

[default = 2]

**void SetRXAmpeakFilEnable (int channel, int fil, int enable)**

channel: number of the receiver channel being accessed

fil: identifier of the filter being enabled. If npeaks is 2, the two filter identifiers will be 0 and 1.

enable: 0 = Disabled; 1 = Enabled

[defaults = { 1 , 1 }]

**void SetRXAmpeakFilFreq (int channel, int fil, double freq)**

channel: number of the receiver channel being accessed

fil: identifier of the filter being enabled. If npeaks is 2, the two filter identifiers will be 0 and 1.

freq: filter center frequency in Hertz

[defaults = { 2125.0, 2295.0 }]

**void SetRXAmpeakFilBw (int channel, int fil, double bw)**

channel: number of the receiver channel being accessed

fil: identifier of the filter being enabled. If npeaks is 2, the two filter identifiers will be 0 and 1.

bw: filter bandwidth in Hertz

[defaults = {75.0, 75.0}]

**void SetRXAmpeakFilGain (int channel, int fil, double gain)**

**channel**: number of the receiver channel being accessed

**fil**: identifier of the filter being enabled. If npeaks is 2, the two filter identifiers will be 0 and 1.

**gain**: gain of the filter. Note that this is an absolute gain --- NOT dB.

[defaults = { 1.0, 1.0 }]

## Patch Panel - Audio Output Configuration

The PatchPanel block is used to configure the audio output. Operations such as volume control, binaural (I/Q) versus monaural output, audio-pan, I/Q swap, etc., can be performed using the provided controls.

### `void SetRXAPanelRun (int channel, int run)`

**channel**: number of the receiver channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 1]

### `void SetRXAPanelSelect (int channel, int select)`

**channel**: number of the receiver channel being accessed

**select**: 0 = no input used; 1 = use only Q input; 2 = use only I input; 3 = Use both I and Q input  
[default = 3]

### `void SetRXAPanelGain1 (int channel, double gain)`

**channel**: number of the receiver channel being accessed

**gain**: gain to be applied to both I and Q outputs. This is an absolute gain, not dB.  
[default = 4.0]

### `void SetRXAPanelGain2 (int channel, double gainI, double gainQ)`

**channel**: number of the receiver channel being accessed

**gainI**: gain to be applied to the I output. This is an absolute gain, not dB.  
[default = 1.0]

**gainQ**: gain to be applied to the Q output. This is an absolute gain, not dB.  
[default = 1.0]

The following call provides an alternative way to set **gainI** and **gainQ**.

### `void SetRXAPanelPan (int channel, double pan)`

**channel**: number of the receiver channel being accessed

**pan**: range is 0.0 to 1.0. If **pan** = 0.0, **gainI** = 1.0 and **gainQ** = 0.0. If **pan** = 1.0, **gainI** = 0.0 and **gainQ** = 1.0. If **pan** = 0.5, **gainI** = 1.0 and **gainQ** = 1.0. There are smooth transitions between these reference points as pan varies from 0.0 to 1.0, smoothly panning the sound.

### `void SetRXAPanelCopy (int channel, int copy)`

**channel**: number of the receiver channel being accessed

**copy**: 0 = no copy; 1 = copy I to Q; 2 = copy Q to I; 3 = reverse I and Q.  
[default = 0]

The following call provides an alternative way to set the '**copy**' flag.

**void SetRXAPanelBinaural (int channel, int bin)**

**channel**: number of the receiver channel being accessed

**bin**: 0 = copy I to Q (monaural mode); 1 = no copy (binaural mode).

## RXA Collectives & General Controls

The following call sets the reception MODE for RXA. This single call turns OFF/ON and provides certain required settings to blocks such as the AM and FM demodulators.

### **void SetRXAMode (int channel, int mode)**

**channel:** number of the receiver channel being accessed

**mode:** mode assignments are: 0 = LSB; 1 = USB; 2 = DSB; 3 = CWL; 4 = CWU; 5 = FM; 6 = AM; 7 = DIGU; 8 = SPEC; 9 = DIGL; 10 = SAM; 11 = DRM.

The following Collective calls combine the calls to multiple RXA blocks in cases where several blocks need the same information. This avoids having the console code make multiple calls, one per block. It also simplifies any future changes where new blocks might be added that require this same information. It is recommended to use these calls rather than individual calls to the blocks when possible.

The upper and lower pass-band frequency limits are set for all RXA blocks needing this information using the call:

### **void RXASetPassband (int channel, double f\_low, double f\_high)**

**channel:** number of the receiver channel being accessed

**f\_low:** lower of the two frequencies (Hertz). May be negative or positive. Must be lower (more negative) than **f\_high**.

**f\_high:** higher of the two frequencies (Hertz). May be negative or positive. Must be higher (more positive) than **f\_low**.

The length of all RXA filter impulse responses for all RXA filters are simultaneously set using the following call. Note that when these filters were created, their 'nc' was set to "max (2048, dsp\_size)". A minimum of 2048 is recommended; although, 1024 may be acceptable in some cases.

### **void RXASetNC (int channel, int nc)**

**channel:** number of the receiver channel being accessed

**nc:** length of the desired filter impulse response. **This must be a power of two AND must be greater than or equal to the DSP Buffer Size in use.**

The choice of Linear Phase or Low Latency for all filters accepting such a choice is set by the call:

### **void RXASetMP (int channel, int mp)**

**channel:** number of the receiver channel being accessed

**mp:** 0 for Linear Phase; 1 for Low Latency (Minimum Phase)

## The TXA Transmitter Unit

### Channel Parameters for TXA

The channel code serves as the interface between the unit it houses and the outside world. Considering this, some of the channel parameters relate only to the outside world. However, others impact the operation or perceived operation of the unit. It is this latter category of parameters that we discuss again here.

**dsp\_size:** The unit only processes data when `dsp_size` samples have been accumulated. This parameter therefore impacts the latency incurred in DSP processing. Generally, using a lower `dsp_size` will reduce latency. However, these lower values mean a larger number of buffers are processed per unit time and each buffer requires some amount of processing overhead. Therefore, lower `dsp_size` implies some increase in the CPU load. **For TXA, for typical `dsp_rate`, `dsp_size` ranging from a minimum of 64 to a maximum of 1024 offers a reasonable range of trade-offs -- 64 for minimum latency and 1024 to minimize CPU load.**

Note, however, that there is an important relationship among `dsp_size`, `dsp_rate`, `in_size`, and `input_samplerate`. For example, consider the case where `dsp_size` = 64, `dsp_rate` = 96000, `input_samplerate` = 48000 and `in_size` = 64. As discussed below, a resampler will be automatically invoked to resample the input from 48000 up to 96000. For each one sample at the 48000 input rate, two samples at the 96000 rate will be produced. Therefore, 64 input samples (`in_size`) will produce 128 samples internally. This is 2 times the 64 samples (`dsp_size`) required for the unit to process data. Therefore, the unit will process twice before the channel will produce output. This implies that making `dsp_size` less than 128 will not speed-up DSP processing, given the other values of this example.

**dsp\_rate:** `dsp_rate` not only impacts latency, as described just above, it also impacts the results in filtering operations. Doubling `dsp_rate` doubles the width of filter transition bands. This can be precisely compensated by doubling the number of filter coefficients. Of course, increasing the rate and increasing the complexity of the filter, both require more CPU cycles. **TXA has been designed to run optimally at a `dsp_rate` of 96000 for all modes.**

**tdelayup, tslewup, tdelaydown, tslewdown:** Typical settings for these parameters are given here. Note that, to minimize CW TX latency, the CW mode is intended to be implemented in firmware rather than through the DSP library. Therefore, lower times for CW are not considered here.

All modes:      `tdelayup` = 0.010; `tslewup` = 0.025; `tdelaydown` = 0.000; `tslewdown` = 0.010.

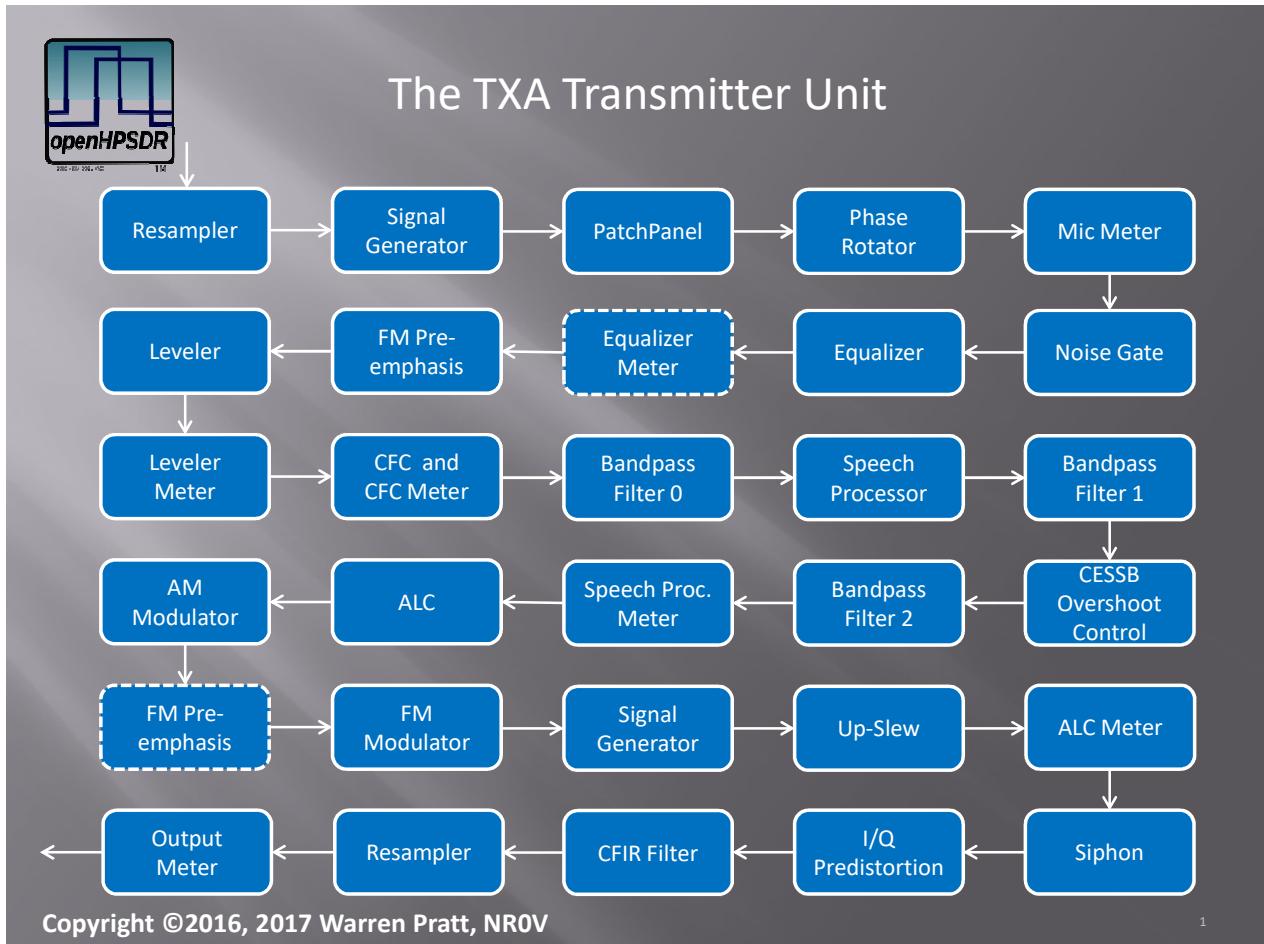
## Signal Characteristics & Levels for TXA

The input to TXA is the audio MIC signal. Although, consistent with other channel input and output signals, this is delivered in an interleaved I-Q packet, this is an audio (NOT an I-Q signal). Hence, only the I values or the Q values will be used. See the "Patchpanel" for selection of whether I or Q data is to be used.

The peak amplitude of the MIC signal is limited only by the range of double-precision floating point calculations. However, it is expected that the product of the peak MIC signal and the MIC Gain setting will be roughly 1.0. See the "Patchpanel" for setting MIC Gain.

The TXA output is an I-Q signal with a peak magnitude of 1.0.

## Block Diagram & Controls



The above figures displays the arrangement of blocks in the TXA transmitter unit. We will now discuss the operation and control of each block.

## **Resamplers**

In the discussion above of channel parameters, three distinct rates are mentioned:

`input_samplerate`: samplerate of data entering the channel

`dsp_rate`: internal rate at which the data is processed within the unit

`output_samplerate`: samplerate of data exiting the channel

Two resamplers are provided within the TXA unit to convert among these rates. The first of these, the "Input Resampler," converts from the `input_samplerate` to the `dsp_rate`. The second of these, the "Output Resampler," converts from the `dsp_rate` to the `output_samplerate`.

No external calls are provided to control these resamplers. The three rates are set by the channel parameters and the resamplers are automatically activated when rates differ and therefore resampling is required.

## Signal Generators

The TXA unit includes two signal generators. The first of these can replace the incoming MIC signal with a variety of waveforms that are useful for testing TXA setup and operation. The second of these generators is near the end of the processing chain and is intended for generating the TUNE signal, a Sweep Carrier, or a Two-Tone Test signal. Calls to set up the first of these generators contain the term "PreGen" while calls to set up the second of these two generators contain the term "PostGen".

The set of output modes currently supported for TXA "PreGen" comprises: Tone, Noise, Sweep, Sawtooth, Triangle, Pulse, and Silence.

### **void SetTXAPreGenRun (int channel, int run)**

**channel:** number of the channel being accessed

**run:** 0 to turn the generator OFF; 1 to turn the generator ON  
[default = 0]

### **void SetTXAPreGenMode (int channel, int mode)**

**channel:** number of the channel being accessed

**mode:** Tone = 0; Noise = 2; Sweep = 3; Sawtooth = 4; Triangle = 5; Pulse = 6; Silence = 99  
[default = 2]

### **void SetTXAPreGenToneMag (int channel, double mag)**

**channel:** number of the channel being accessed

**mag:** peak magnitude value of the single-tone (mode = 0). Typical range is 0.0 to 1.0.  
[default = 1.0]

### **void SetTXAPreGenToneFreq (int channel, double freq)**

**channel:** number of the channel being accessed

**freq:** frequency (Hertz) of the single-tone (mode = 0). Can be either positive or negative.  
[default = +1000.0]

### **void SetTXAPreGenNoiseMag (int channel, double mag)**

**channel:** number of the channel being accessed

**mag:** magnitude value of the noise (mode = 2). Typical range is 0.0 to 1.0.  
[default = 1.0]

### **void SetTXAPreGenSweepMag (int channel, double mag)**

**channel:** number of the channel being accessed

**mag:** magnitude of the swept single tone (mode = 3). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetTXAPreGenSweepFreq (int channel, double freq1, double freq2)**

**channel:** number of the channel being accessed

**freq1, freq2:** frequency limits of the sweep range (Hertz). Sweep progresses from freq1 to freq2 and then repeats.

[defaults: freq1 = -20000.0, freq2 = +20000.0]

**void SetTXAPreGenSweepRate (int channel, double rate)**

**channel:** number of the channel being accessed

**rate:** rate at which the sweep progresses (Hertz / Second).

[default = +4000.0]

**void SetTXAPreGenSawtoothMag (int channel, double mag)**

**channel:** number of the channel being accessed

**mag:** peak magnitude value of the sawtooth. Typical range is 0.0 to 1.0.

[default = 1.0]

**void SetTXAPreGenSawtoothFreq (int channel, double freq)**

**channel:** number of the channel being accessed

**freq:** frequency (Hertz) of the sawtooth.

[default = +500.0]

**void SetTXAPreGenTriangleMag (int channel, double mag)**

**channel:** number of the channel being accessed

**mag:** peak magnitude value of the triangle. Typical range is 0.0 to 1.0.

[default = 1.0]

**void SetTXAPreGenTriangleFreq (int channel, double freq)**

**channel:** number of the channel being accessed

**freq:** frequency (Hertz) of the triangle.

[default = +500.0]

Note that the "Pulse" generator mode can produce a pulsed-tone, really a "burst mode", if the tone frequency is set to a non-zero value.

**void SetTXAPreGenPulseMag (int channel, double mag)**

channel: number of the channel being accessed

mag: peak magnitude value of the pulse. Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetTXAPreGenPulseFreq (int channel, double freq)**

channel: number of the channel being accessed

freq: frequency (Hertz) of occurrence of the pulse.  
[default = 0.25]

**void SetTXAPreGenPulseDutyCycle (int channel, double dc)**

channel: number of the channel being accessed

dc: Duty-cycle of the pulse = fraction of the time that the pulse is ON.  
[default = 0.25]

**void SetTXAPreGenPulseToneFreq (int channel, double freq)**

channel: number of the channel being accessed

freq: Frequency of a pulsed tone (Hertz).  
[default = 1000.0]

**void SetTXAPreGenPulseTransition (int channel, double transtime)**

channel: number of the channel being accessed

transtime: OFF-ON or ON-OFF transition time (seconds).  
[default = 0.002]

The set of output modes currently supported for TXA "PostGen" comprises: Tone, Two-tone, Sweep, and Silence.

**void SetTXAPostGenRun (int channel, int run)**

channel: number of the channel being accessed

run: 0 to turn the generator OFF; 1 to turn the generator ON  
[default = 0]

**void SetTXAPostGenMode (int channel, int mode)**

channel: number of the channel being accessed

**mode**: Tone = 0; Two-Tone = 1; Sweep = 3; Pulsed tone = 6; Pulsed two-tone = 7; Silence = 99.  
[default = 0]

**void SetTXAPostGenToneMag (int channel, double mag)**

**channel**: number of the channel being accessed

**mag**: peak magnitude value of the single-tone (mode = 0). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetTXAPostGenToneFreq (int channel, double freq)**

**channel**: number of the channel being accessed

**freq**: frequency (Hertz) of the single-tone (mode = 0). Can be either positive or negative.  
[default = +1000.0]

**void SetTXAPostGenTTMag (int channel, double mag1, double mag2)**

**channel**: number of the channel being accessed

**mag1**: peak magnitude value of the first tone (mode = 0). Typical range is 0.0 to 1.0.  
[default = 0.5]

**mag2**: peak magnitude value of the second tone (mode = 0). Typical range is 0.0 to 1.0.  
[default = 0.5]

NOTE: With both **mag1** and **mag2** set to 0.5, the peak magnitude of the generator output will be 1.0. I.e., the peak magnitude of the generator output is **mag1 + mag2**.

**void SetTXAPostGenTTFreq (int channel, double freq1, double freq2)**

**channel**: number of the channel being accessed

**freq1**: frequency (Hertz) of the first tone (mode = 0). Can be either positive or negative.  
[default = + 900.0]

**freq2**: frequency (Hertz) of the second tone (mode = 0). Can be either positive or negative.  
[default = +1700.0]

**void SetTXAPostGenSweepMag (int channel, double mag)**

**channel**: number of the channel being accessed

**mag**: magnitude of swept single tone (mode = 3). Typical range is 0.0 to 1.0.  
[default = 1.0]

**void SetTXAPostGenSweepFreq (int channel, double freq1, double freq2)**

**channel**: number of the channel being accessed

**freq1, freq2**: frequency limits of the sweep range (Hertz). Sweep progresses from freq1 to freq2 and then repeats.

[defaults: freq1 = -20000.0, freq2 = +20000.0]

**void SetTXAPostGenSweepRate (int channel, double rate)**

**channel**: number of the channel being accessed

**rate**: rate at which the sweep progresses (Hertz / Second).

[default = +4000.0]

**void SetTXAPostGenPulseMag (int channel, double mag)**

**channel**: number of the channel being accessed

**mag**: magnitude of pulsed single tone (mode = 6). Range is 0.0 to 1.0.

[default = 1.0]

**void SetTXAPostGenPulseFreq (int channel, double freq)**

**channel**: number of the channel being accessed

**freq**: pulsing frequency (not tone frequency) of pulsed single tone (mode = 6), in Hertz.

[default = 2.0 (Hz)]

**void SetTXAPostGenPulseDutyCycle (int channel, double dc)**

**channel**: number of the channel being accessed

**dc**: duty-cycle of pulse (mode = 6),  $0.0 < dc < 1.0$ .

[default = 0.25]

**void SetTXAPostGenPulseToneFreq (int channel, double freq)**

**channel**: number of the channel being accessed

**freq**: tone frequency in Hertz. (mode = 6).

[default = 600.0]

**void SetTXAPostGenPulseTransition (int channel, double transtime)**

**channel**: number of the channel being accessed

**transtime**: time used for a pulse transition from OFF to ON or from ON to OFF, in seconds. This is a raised-cosine transition. (mode = 6).

[default = 0.005]

**void SetTXAPostGenPulseIQout (int channel, int IQout)**

**channel**: number of the channel being accessed

**IQout**: specifies either audio output (I only, Q = 0) or I-Q output ('0' => I-only; '1' => IQ). (mode = 6). For use in the TX chain, 'IQout' should normally be set to '1'  
[default = 0]

---

**void SetTXAPostGenTTPulseMag (int channel, double mag1, double mag2)**

**channel**: number of the channel being accessed

**mag1, mag2**: magnitude of each of the two tones (mode = 7). Range for each is 0.0 to 1.0. However, the sum of the two should not exceed 1.0!  
[default = 0.5, 0.5]

**void SetTXAPostGenTTPulseFreq (int channel, double freq)**

**channel**: number of the channel being accessed

**freq**: pulsing frequency (not tone frequencies) of pulsed two-tone (mode = 7), in Hertz.  
[default = 2.0 (Hz)]

**void SetTXAPostGenTTPulseDutyCycle (int channel, double dc)**

**channel**: number of the channel being accessed

**dc**: duty-cycle of pulse (mode = 7),  $0.0 < dc < 1.0$ .  
[default = 0.25]

**void SetTXAPostGenTTPulseToneFreq (int channel, double freq1, double freq2)**

**channel**: number of the channel being accessed

**freq1, freq2**: tone frequencies in Hertz. (mode = 7).  
[default = 900.0, 1700.0]

**void SetTXAPostGenTTPulseTransition (int channel, double transtime)**

**channel**: number of the channel being accessed

**f**: time used for a pulse transition from OFF to ON or from ON to OFF, in seconds. This is a raised-cosine transition. (mode = 7).  
[default = 0.005]

**void SetTXAPostGenTTPulseIQout (int channel, int IQout)**

**channel**: number of the channel being accessed

**IQout**: specifies either audio output (I only, Q = 0) or I-Q output ('0' => I-only; '1' => IQ). (mode = 7). For use in the TX chain, 'IQout' should normally be set to '1'!

[default = 0]

## PatchPanel

The TXA patchpanel is used to select which channel (I or Q) is used for MIC input and to set the MIC gain.  
Available calls are:

**void SetTXAPanelRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = this patchpanel is not used; 1 = this patchpanel is active  
[default = 1]

**void SetTXAPanelSelect (int channel, int select)**

**channel**: number of the channel being accessed

**select**: 1 for Q; 2 for I  
[default = 2]

**void SetTXAPanelGain1 (int channel, double gain)**

**channel**: number of the channel being accessed

**gain**: MIC Gain to be applied (this is a linear value of gain, NOT a dB value)  
[default = 1.0]

## Meters

As shown in the block diagram, metering outputs from TXA are the Mic Meter, Equalizer Meter, Leveler Meter, CFC Meter, Speech Processor Meter, ALC Meter, and the Output Meter. Taken together, these provide visibility of internal TXA processing levels that is sufficient to optimize many TXA settings.

Meter values are retrieved using the single call:

**double GetTXAMeter (int channel, int mt)**

**return value:** requested meter value (dBV, except dB for Leveler\_Gain and ALC\_Gain). This will return a value of 0.0 dBV for a signal magnitude level of 1.0.

**channel:** number of the channel being accessed

**mt:** type of meter data being requested. **mt** values are:

MIC\_Peak = 0;

MIC\_Average = 1;

EQ\_Peak = 2;

EQ\_Average = 3;

Leveler\_Peak = 4;

Leveler\_Average = 5;

Leveler\_Gain = 6;

CFC\_Peak = 7;

CFC\_AV = 8;

CFC\_Gain = 9;

Compressor\_Peak = 10;

Compressor\_Average = 11;

ALC\_Peak = 12;

ALC\_Average = 13;

ALC\_Gain = 14;

Output\_Peak = 15;

Output\_Average = 16;

## Noise Gate

The Noise Gate can be set to lower the microphone audio gain if the input signal for a period of time is below a specified level. This can be used to lower transmitted background noise (for example fan noise) during pauses in speech.

An instance of the same "AM Squelch" used in the RXA receiver is used for this purpose. Therefore, many of the same features, like raised-cosine transitions between gain levels, are available.

To activate this feature:

```
void SetTXAAMSQRun (int channel, int run)
```

**channel**: number of the channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 0]

To set the gain when "muted", i.e., the low gain level:

```
void SetTXAAMSQMutedGain (int channel, double dBlevel)
```

**channel**: number of the channel being accessed

**dBlevel**: gain (dB) to be used when the noise gate is MUTED. This is a negative value, relative to the UN-MUTED level.  
[default = -13.98 dB]

To set the threshold for "un-mute":

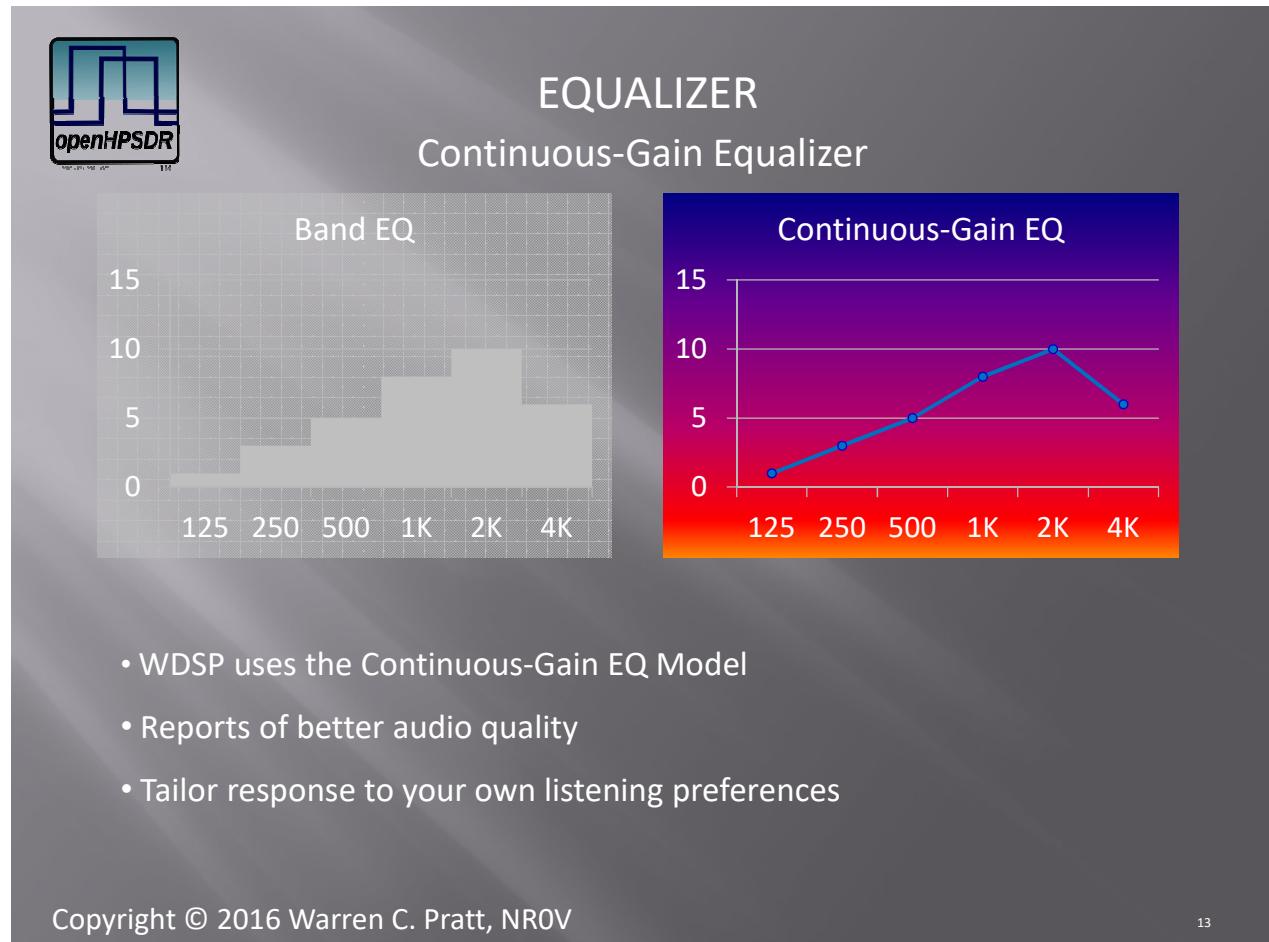
```
void SetTXAAMSQThreshold (int channel, double threshold)
```

**channel**: number of the channel being accessed

**threshold**: threshold (dB) at which to un-mute the noise gate.  
[default = -13.98 dB]

## Equalizer

The equalizer uses a continuous frequency response algorithm meaning that the gain is specified at any number of frequencies and the response is interpolated between the specified points. This is in contrast to a frequency "band" approach.



The equalizer is turned OFF/ON using calls to:

```
void SetTXAEQRun (int channel, int run)
```

**channel**: number of the channel being accessed

**run**: 0 for OFF; 1 for ON  
[default = 0]

The length of the EQ filter impulse response can be set with the following call. However, this is **NOT RECOMMENDED**. The preferred method is to use the TXA Collective **TXASetNC(...)**. (See the section of this document on "TXA Collectives".)

```
void SetTXAEQNC (int channel, int nc)
```

**DEPRECATED**

**channel**: number of the channel being accessed

**nc**: length of the EQ filter impulse response. 'nc' must be a power-of-two AND must be greater than or equal to the DSP Buffer Size in use.

[default = 2048]

The Filter Type (Linear-Phase / Low-Latency) for the EQ filter can be set with the following call.

However, this is **NOT RECOMMENDED**. The preferred method is to use the TXA Collective **TXASetMP(...)**. (See the section of this document on "TXA Collectives".)

**void SetTXAEQMP (int channel, int mp)**

**DEPRECATED**

**channel**: number of the channel being accessed

**mp**: 0 for Linear-Phase; 1 for Low-Latency (Minimum Phase)

[default = 0]

The window-type used in the generation of the EQ filter can be selected. This setting should normally just be left at the default value.

**void SetTXAEQWintype (int channel, int wintype)**

**channel**: number of the channel being accessed

**wintype**: 0 = 4-term Blackman-Harris; 1 = 7-term Blackman-Harris

[default = 0]

The cutoff-mode used in the generation of the EQ filter can be selected. This setting should normally just be left at the default value.

**void SetTXAEQCtfmode (int channel, int mode)**

**channel**: number of the channel being accessed

**mode**: 0 = 4<sup>th</sup>-power roll-off on each side of the specified passband; 1 = no roll-off outside the specified passband

[default = 0]

The following call is the primary and preferred method of setting the equalizer response.

**void SetTXAEQProfile (int channel, int nfreqs, double\* F, double\* G)**

**channel**: number of the channel being accessed

**nfreqs**: number of frequency/gain pairs to be specified

**F**: pointer to an array of frequencies (Hertz). NOTE: The 0<sup>th</sup> element of this vector is NOT USED. So, for example, if **nfreqs** is 5, F[0] is not used and the five frequencies will be stored in F[1] ... F[5]. The F[] array has (**nfreqs** + 1) values.

Note that acceptable values are between 0.0 Hz and **dsp\_rate**/2.0 Hz. It is acceptable to specify the same frequency multiple times; however, if different gains are specified for the same frequency, it is indeterminate as to which of the gains will be applied at that frequency. Frequency/Gain pairs do not necessarily have to be specified in order of increasing frequency. If they are not, they will be sorted into the appropriate order prior to application.

**G**: pointer to a corresponding array of gain values (dB). The 0<sup>th</sup> element is used to specify a "preamp gain" value to be applied to all frequencies. The G[] array has (**nfreqs** + 1) values.

For convenience and consistency with legacy consoles, two "short-cut" calls with pre-specified frequencies are provided.

**void SetTXAGrphEQ (int channel, int\* rxeq)**

**DEPRECATED**

**channel**: number of the channel being accessed

**rxeq**: pointer to an array containing gain values (dB) corresponding to an implicit "F vector" of F[] = {0.0, 150.0, 400.0, 1500.0, 6000.0}. The value in G[0] is the "preamp gain" to be applied to all frequencies. G[1] through G[4] contain gains corresponding to 150.0 Hz through 6000.0 Hz, respectively.

**void SetTXAGrphEQ10 (int channel, int\* rxeq)**

**DEPRECATED**

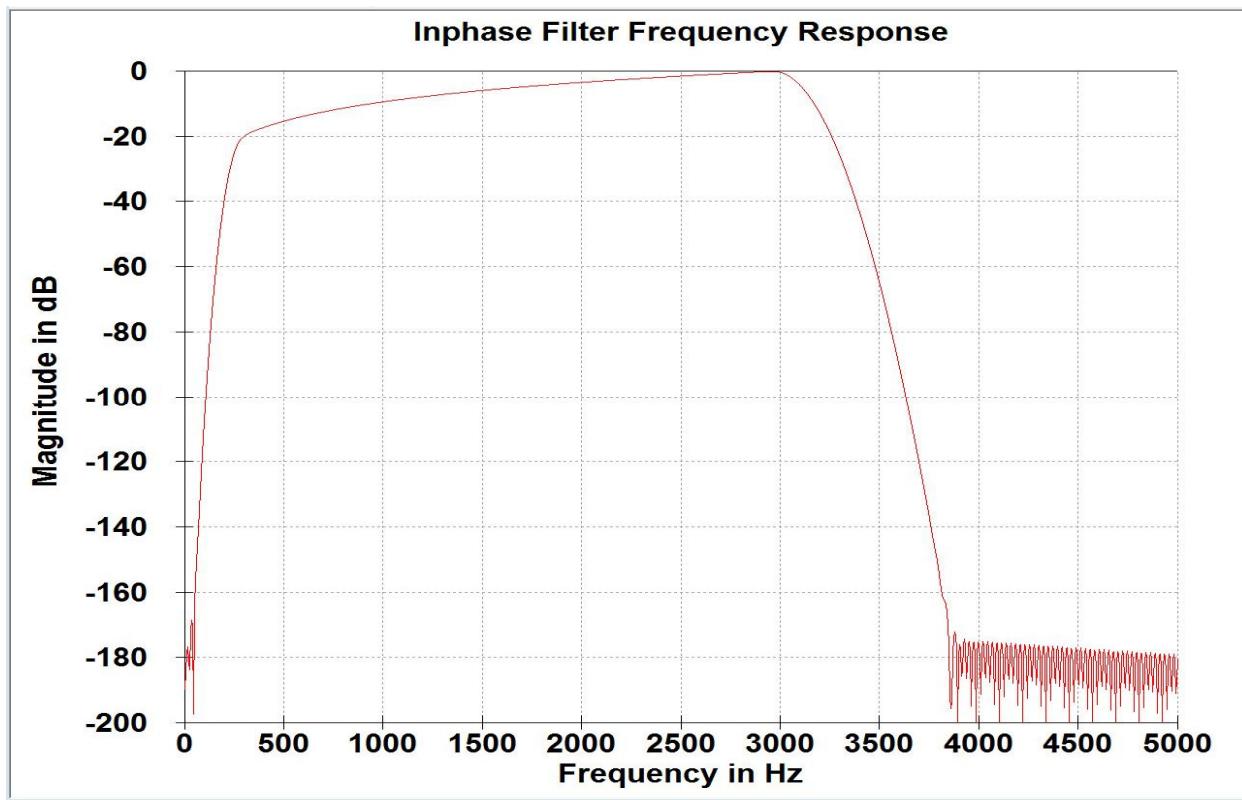
**channel**: number of the channel being accessed

**rxeq**: pointer to an array containing gain values (dB) corresponding to an implicit "F vector" of F[] = {0.0, 32.0, 63.0, 125.0, 250.0, 500.0, 1000.0, 2000.0, 4000.0, 8000.0, 16000.0}. The value in G[0] is the "preamp gain" to be applied to all frequencies. G[1] through G[10] contain gains corresponding to 32.0 Hz through 16000.0 Hz, respectively.

## FM Pre-emphasis

FM transmit Pre-emphasis, along with companion receive de-emphasis, are used to improve the signal-to-noise ratio of FM transmission-reception.

The standard bandwidth for narrowband FM communications is 300 Hz to 3000 Hz. Pre-emphasis follows a 6 dB per octave (20 dB per decade) slope. When implemented with the default impulse response length (2048), the TXA pre-emphasis curve is as shown in the following figure. Note that the frequency scale shown here is linear; therefore the curve appears to be more rounded than the straight-line one would see if this were plotted with a log-domain frequency axis. Also, note that no attempt has been made here to make the cutoff extremely sharp as this will be controlled by the bandpass filter.



Referring back to the overall TXA block diagram, note that FM Pre-emphasis is shown twice, each time in a block with a dashed outside line. The meaning is that the FM pre-emphasis can be located in either position, controllable by a "switch":

```
void SetTXAFMEmphPosition (int channel, int position)
```

**channel:** number of the channel being accessed

**position:** 0 = BEFORE pre-processing and ALC; 1 = AFTER pre-processing and ALC  
[default = 1]

The latter (default) position will absolutely limit the transmitted magnitude of a particular frequency, i.e., a 1000 Hz could never be transmitted with the same power as a 2000 Hz signal. The former position

has a different behavior which generally may allow the signal to sound louder. The correct positioning has been the subject of some discussion and so both options are provided via the switch.

The length of the Pre-emphasis filter impulse response can be set with the following call. However, this is **NOT RECOMMENDED**. The preferred method is to use the TXA Collective [TXASetNC\(...\)](#).

**void SetTXAFMEmphNC (int channel, int nc)**

**DEPRECATED**

**channel:** number of the channel being accessed

**nc:** length of the Pre-emphasis filter impulse response. 'nc' must be a power-of-two AND must be greater than or equal to the DSP Buffer Size in use.

[default = 2048]

The Filter Type (Linear-Phase / Low-Latency) for the Pre-emphasis filter can be set with the following call. However, this is **NOT RECOMMENDED**. The preferred method is to use the TXA Collective [TXASetMP\(...\)](#).

**void SetTxAEmphMP (int channel, int mp)**

**DEPRECATED**

**channel:** number of the channel being accessed

**mp:** 0 for Linear-Phase; 1 for Low-Latency (Minimum Phase)

[default = 0]

## Leveler

The audio leveler can be invoked to provide additional gain when the audio signal is low and then to reduce gain as the audio level increases. This can be useful, for example, to provide a little extra gain if the operator momentarily turns away from the microphone or to reduce gain if the operator momentarily gets closer to the microphone, in other words, it "levels" the audio signal.

The Leveler also helps provide a known audio amplitude level for the phase rotator, CFC, and speech processor (compressor) that follow. This helps ensure that the requested compression level is the level delivered. In the absence of this assistance from the Leveler, it's solely up to the operator to ensure that the audio level before the following stages is peaking at about 0 dB.

The Leveler is implemented with an instance of AGC/ALC code; however, the settings are considerably different for the Leveler application. You will, however, notice similarity in some of the settings. Settings available for the Leveler are:

**void SetTXALevelerSt (int channel, int state)**

**channel**: number of the channel being accessed

**state**: 0 = OFF; 1 = ON  
[default = 0]

**void SetTXALevelerAttack (int channel, int attack)**

**channel**: number of the channel being accessed

**attack**: attack time-constant in ms. Note that this is a time-constant for an exponential curve, NOT an absolute time. It is recommended to leave this at the default value.  
[default = 1 ms]

**void SetTXALevelerDecay (int channel, int decay)**

**channel**: number of the channel being accessed

**decay**: decay time-constant in ms. Note that this is a time-constant for an exponential curve, NOT an absolute time. The default value works well for normal operating conditions.  
[default = 500 ms]

**void SetTXALevelerTop (int channel, double maxgain)**

**channel**: number of the channel being accessed

**maxgain**: maximum gain (dB) that the leveler is to apply.  
[default = 5.0 dB]

## Phase Rotator

In some AM transmitters, it is possible to boost the peak output power by having an asymmetrical audio waveform (positive peaks greater than negative peaks) and modulating to greater than 100% on positive peaks while restricting to <=100% on negative peaks to avoid "pinch-off."

In other AM transmitters and in the case of our digital-up-conversion (DUC) SDRs, this is not the correct approach. We have hard-limits which cannot be exceeded, such as the dynamic range of the DAC. The application of an asymmetrical audio waveform, with positive peaks greater than negative peaks, cannot further increase modulation in the positive direction; it instead REDUCES average power. The correct approach in such cases is to make the audio waveform as symmetrical as possible, i.e., equal positive and negative peaks.

The Phase Rotator makes the audio waveform more symmetrical. It does so by shifting the phase of various audio frequencies by varying amounts, thereby changing the shape of the waveform away from the somewhat typical asymmetrical waveform of human speech. Experimentation and analysis show that a wide range of phase shift versus frequency generally tends to improve symmetry. This wide range leads to an implementation with multiple identical stages, where the total phase shift is the sum of the shifts obtained in each stage. The stages have a specified "corner frequency" where the phase shift of the stage is equal to one-half of the total that the stage provides at the maximum frequency.

While the above explanation focused on AM, note that this feature can be used to increase average power for any speech mode. Note also, however, that it could be detrimental for any digital mode that requires coherent phase versus frequency.

Settings for the Phase Rotator are:

**void SetTXAPHROTRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 0]

**void SetTXAPHROTCorner (int channel, double frequency)**

**channel**: number of the channel being accessed

**frequency**: Corner frequency in Hertz.  
[default = 338.0]

**void SetTXAPHROTNstages (int channel, int nstages)**

**channel**: number of the channel being accessed

**nstages**: The number of phase rotator stages to be used.  
[default = 8]

Also included in the phase rotator is a function that can be used to reverse microphone phase, i.e., shift it by 180 degrees.

**void SetTXAPHROTReverse (int channel, int reverse)**

**channel**: number of the channel being accessed

**reverse**: 0 = OFF; 1 = ON

[default = 0]

## Continuous Frequency Compressor (CFC)

Many audio racks and Digital Audio Workstations provide "Multiband Compressors" which allow specifying different amounts of compression for different audio frequency bands. The CFC offers a superset of that concept where, instead of utilizing multiple bands with constant compression in each, the compression varies smoothly between frequency points at which it is specified. This concept is similar to the WDSP Equalizer function discussed above; however, this time, we are varying the compression level across frequencies rather than gain. A Post-CFC Equalizer is also provided as an integral part of this function.

Use of this function for speech modes can significantly increase the "density" and average power of the signal. Note also that use for digital modes may be detrimental, depending upon the nature of the mode.

Settings for the CFC are:

**void SetTXACFCOMPRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 0]

**void SetTXACFCOMPprofile (int channel, int nfreqs, double\* F, double\* G, double\* E)**

**channel**: number of the channel being accessed

**nfreqs**: number of frequencies at which compression and post-CFC equalizer gain are specified

**F**: pointer to the array of frequency points (values in Hertz)

**G**: pointer to the array of compression at each frequency point (values in dB)

**E**: pointer to the array of post-CFC equalizer gains (values in dB)

**void SetTXACFCOMPPrecomp (int channel, double precomp)**

**channel**: number of the channel being accessed

**precomp**: an additional amount of compression to be applied equally across the frequency spectrum (dB) [default = 0.0]

**void SetTXACFCOMPPeqRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: Turn OFF/ON the post-CFC Equalizer: 0 = OFF; 1 = ON [default = 0]

**void SetTXACFCOMPPrePeq (int channel, double prepeq)**

**channel**: number of the channel being accessed

**preeq**: an additional amount of gain to be applied equally across the frequency spectrum in the post-CFC equalizer (dB) [default = 0.0]

If it is desired to display the contour of compression versus frequency for adjustment purposes, a function has been provided to supply that information.

**void GetTXACFCOMPDisplayCompression (int channel, double\* comp\_values, int\* ready)**

**channel**: number of the channel being accessed

**comp\_values**: pointer to storage for compression values (in dB) for each of the FFT bins. As currently set up, this will return 1025 values.

**ready**: pointer to storage for a flag indicating whether or not new values are available since the last call was made to this function. If no new values are available, there would be no reason to re-draw the compression versus frequency display.

## Bandpass Filters

There are three bandpass filters:

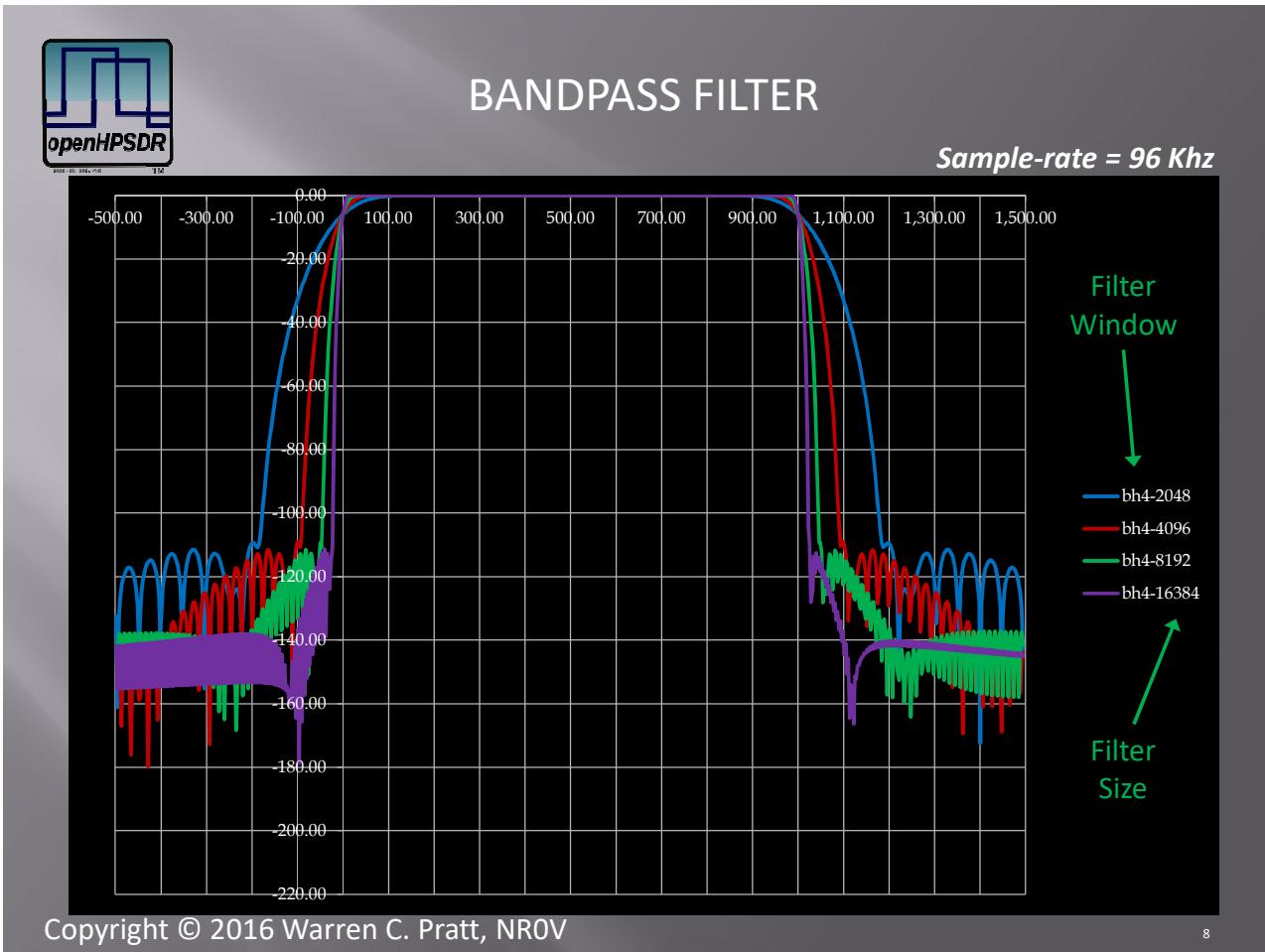
- The first of these, Bandpass Filter 0, is always ON. It "squares up" the pass-band and converts the audio to a complex I-Q signal. This is the primary bandpass filter.
- The second, Bandpass Filter 1, is used if and only if the Speech Processor is used. The speech processor is a non-linear device and therefore creates out-of-band frequency components. This bandpass filter removes those components. This filter is also a prerequisite for the CESSB Overshoot Control. The ON/OFF state of this filter is automatically set inside WDSP.
- The third, Bandpass Filter 2, is used if and only if the CESSB Overshoot Control is activated and serves as the final overshoot correction step for this algorithm. The ON/OFF state is automatically set inside WDSP.

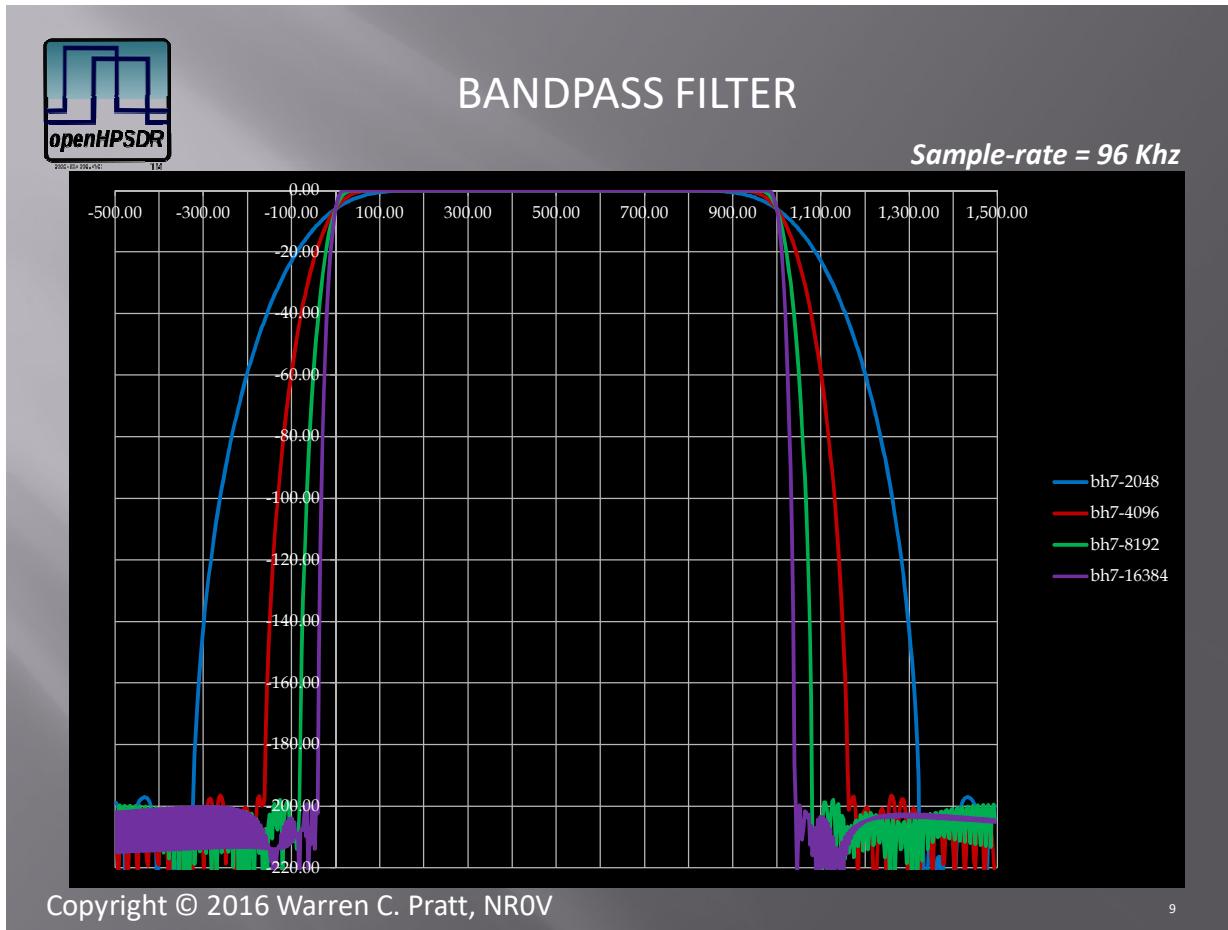
For AM, AM\_LSB, AM\_USB, FM, and DSB modes, it is expected that the filter frequencies will be set by the console to cover an equal-sided range, e.g.,  $f_{low} = -5000.0$  and  $f_{high} = +5000.0$ . Any other necessary adjustments are handled internally in the WDSP code.

For other single-sided modes, such as LSB and USB, filter frequencies should be set as desired for the mode. For example, for LSB,  $f_{low} = -3000.0$  and  $f_{high} = -200.0$ .

The "sharpness" of these filters, when running TXA at a `dsp_rate` of 96000, is shown in the following two figures. The first of these figures shows the response curves for different impulse response lengths, all using the 4-term Blackman-Harris window. The second figure is the same except using the 7-term Blackman-Harris window. The 4-term yields slightly sharper cutoff; however, less cutoff depth. The 4-term would be preferred in most situations since its cutoff depth is generally more than sufficient; the 7-term is overkill on cutoff depth for most situations.

Note that the transition bands (regions between the pass-band and the cutoff bands) are the most important part to observe in these figures. These transition bands will maintain the same width independent of the width of the filter pass-band.





The lower and upper frequency cutoffs for the three bandpass filters are set using the call:

**void SetTXABandpassFreqs (int channel, double f\_low, double f\_high)**

**channel**: number of the channel being accessed

**f\_low**: low cutoff frequency (Hz). May be either positive or negative. Must be less (more negative) than **f\_high**.

[default = -5000.0 Hz]

**f\_high**: high cutoff frequency (Hz). May be either positive or negative. Must be greater (more positive) than **f\_low**.

[default = -100.0 Hz]

The window function to be used in generating the bandpass filters is set using:

**void SetTXABandpassWindow (int channel, int wintype)**

**channel**: number of the channel being accessed

**wintype**: 0 = Blackman-Harris 4-term; 1 = Blackman-Harris 7-term

The length of the bandpass filters' impulse responses could be set using the following call. However, this is **NOT RECOMMENDED**. Instead, it is recommended to use the TXA Collective call [TXASetNC\(...\)](#). (See the section of this document on TXA Collectives.)

**void SetTXABandpassNC (int channel, int nc)**

**DEPRECATED**

**channel:** number of the channel being accessed

**nc:** length of the filter impulse response. Must be a power of two and must be greater than or equal to the DSP buffer size in use.

[default = 2048]

The choice of Linear Phase or Low Latency for the bandpass filters could be set using the following call. However, this is **NOT RECOMMENDED**. Instead, it is recommended to use the TXA Collective call [TXASetMP\(...\)](#). (See the section of this document on TXA Collectives.)

**void SetTXABandpassMP (int channel, int mp)**

**DEPRECATED**

**channel:** number of the channel being accessed

**mp:** 0 = Linear Phase; 1 = Low Latency (Minimum Phase)

## Speech Processor

The Speech Processor is the equivalent of an RF Speech Clipper; however, implemented at baseband rather than at RF frequencies.

Note that the specified compression level is accurate only when the input signal is achieving a peak magnitude of exactly 1.0. At lower peak input levels, less compression will be achieved and at higher peak input levels more compression will be achieved. As previously discussed (see "Leveler"), the Leveler can provide assistance in regulating the input level. Whether the Leveler is used or not, the metering functionality provides excellent feedback to the operator concerning the signal levels in the audio chain.

The Speech Processor is turned OFF/ON using the following call:

**void SetTXACompressorRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 0]

The compression level is set using:

**void SetTXACompressorGain (int channel, double gain)**

**channel**: number of the channel being accessed

**gain**: compression level in dB.  
[default = 3.0 dB]

## CESSB Overshoot Control

The concept of CESSB (Controlled Envelope Single Sideband) was provided to the Amateur Radio community by Dave Hershberger, W9GR.<sup>2</sup> In brief, this is an algorithm to increase the average transmitted power of an SSB signal while introducing only minimal distortion. To understand more detail, please see Dave's article.

Only one control is needed, the OFF/ON control:

**void SetTXAosctrlRun (int channel, int run)**

**channel:** number of the channel being accessed

**run:** 0 = OFF; 1 = ON  
[default = 0]

HOWEVER, note that it is the responsibility of the console application to ensure that the Speech Processor is turned ON as a prerequisite for this block! This block will NOT operate properly in the absence of the Speech Processor also being active!

Note that for optimum performance, Linear Phase bandpass filters should be used with this feature. The amount of performance degradation when using Low Latency filters has not been measured and will vary with voice characteristics.

---

<sup>2</sup> David L. Hershberger, W9GR, "Controlled Envelope Single Sideband," QEX Magazine, November/December 2014.

## ALC

Unlike conventional radios, the Automatic Level Control function for TXA is accomplished in a single block; it does not include any kind of feedback from later blocks in the pipeline. ALC systems with feedback loops generally "drive by looking in the rear-view mirror," i.e., they react to an over-drive event in a later stage by reducing the gain in an earlier stage. However, that gain reduction is after-the-fact ... the over-drive and initial distortion have already occurred. The ALC system in TXA contains a short delay line which allows it to "see" events coming on the horizon before it has to provide output for them. Therefore, it can gradually reduce the gain along an exponential curve BEFORE the event happens rather than after.

In a Digital Up-Conversion (DUC) SDR, it is EXTREMELY important to insure that peak magnitudes NEVER exceed a specific level. Even though larger values would be no problem in WDSP floating-point calculations, they would overflow the (finite-range, integer-input) DAC, and, perhaps the integer calculations preceding the DAC. These overflows produce extreme distortion and very broadband splatter. The WDSP ALC system is designed to avoid this happening. For this reason, the ALC should ALWAYS be active.

Also, per this type ALC action, there is no ALC overshoot produced by this system.

For testing purposes, the ALC can be turned OFF/ON using the following call. HOWEVER, per the discussion above, it is expected that the ALC will ALWAYS BE ON.

**void SetTXAALCSt (int channel, int state)**

**channel:** number of the channel being accessed

**state:** 0 = OFF; 1 = ON

[default = 1]

The ALC Attack time-constant can be set using the following call. It is recommended to leave this at the default value.

**void SetTXAALCAttack (int channel, int attack)**

**channel:** number of the channel being accessed

**attack:** attack time-constant (ms). Note that this is a time-constant for an exponential curve, not an absolute time.

[default = 1 ms]

The ALC Decay time-constant is set using the following call.

**void SetTXAALCDecay (int channel, int decay)**

**channel:** number of the channel being accessed

**decay**: decay time-constant (ms). Note that this is a time-constant for an exponential curve, not an absolute time.

[default = 10 ms]

A maximum gain for the ALC block is specified with the following call. This is "maximum gain" since the ALC will reduce the gain (compress) as needed to deliver the targeted output level.

**void SetTXAALCMaxGain (int channel, double maxgain)**

**channel**: number of the channel being accessed

**maxgain**: maximum gain (dB)

[default = 0dB]

## AM Modulator

The AM Modulator is automatically activated/deactivated based upon the [SetTXAMode\(...\)](#) setting.  
(See the section on "TXA Collectives & General Controls.")

The only setting for the AM Modulator is the Carrier Level setting:

```
void SetTXAAMCarrierLevel (int channel, double c_level)
```

**channel**: number of the channel being accessed

**c\_level**: 0.0 => NO CARRIER (DSB); 0.5 => 100% modulation; >0.5 => less than 100% modulation.  
[default = 0.5]

Note that the carrier level setting applies to the modes AM\_LSB and AM\_USB as well as the AM mode.

## FM Modulator

The FM Modulator is automatically activated/deactivated based upon the [SetTXAMode\(...\)](#) setting. (See the section on "TXA Collectives & General Controls.")

The peak deviation for the FM signal can be set using the call:

**void SetTXAFMDeviation (int channel, double deviation)**

**channel**: number of the channel being accessed

**deviation**: peak deviation of the FM modulation (Hz).

[default = 5000.0]

A CTCSS tone can optionally be transmitted on the FM signal. There are controls to turn the tone OFF/ON and to set its frequency.

**void SetTXACTCSSRun (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = tone OFF; 1 = tone ON.

[default = 1]

**void SetTXACTCSSFreq (int channel, double freq)**

**channel**: number of the channel being accessed

**freq**: tone frequency (Hz).

[default = 100.0]

The highest audio modulating frequency for the FM mode is 3000 Hz. Carson's Rule states that about 98% of the power of the resulting FM signal is contained within a bandwidth ranging from minus(deviation+highest\_audio\_frequency) to plus(deviation+highest\_audio\_frequency). While, at first glance, that may seem significant, that means that the power outside this range is only 17 dB below the power within the range. Therefore, the FM Modulator includes an output bandpass filter that limits the transmitted power to be within this specified range.

The length of the impulse response of this filter, and whether it is Linear Phase or Low Latency, can be set using the following calls. This is **NOT RECOMMENDED**. Instead, it is recommended to use the calls [TXASetNC\(...\)](#) and [TXASetMP\(...\)](#), respectively, found in the TXA Collectives section.

**void SetTXAFMNC (int channel, int nc)**

**DEPRECATED**

**channel**: number of the channel being accessed

**nc**: length of the filter impulse response. This must be a power-of-two and must be greater than or equal to the **dsp\_size** in use at the time.

[default = 2048]

**void SetTXAFMMP (int channel, int mp)**

**DEPRECATED**

**channel**: number of the channel being accessed

**mp**: 0 = Linear Phase; 1 = Low Latency (Minimum Phase).

The normal audio frequency range for narrow-band FM communication is 300 Hz to 3000 Hz. This can be adjusted with the call:

**void SetTXAFMAFFilter (int channel, double low, double high)**

**channel**: number of the channel being accessed

**low**: low-cut frequency (Hz)

**high**: high-cut frequency (Hz)

However, when setting these frequencies to wider limits, it is also necessary to open-up other TX bandpass limits which can be done with a call to **SetTXABandpassFreqs ( ... )**. Note also that when going lower in frequency, you'll may be overlapping the range in which CTCSS tones are transmitted. When going higher in frequency, you will be increasing your transmit bandwidth which, by Carson's Rule is equal to  $2 * (\text{maximum\_deviation} + \text{highest\_audio\_frequency})$ .

## Up-Slew

Note that the Up-Slew block is located after (1) the Signal Generators, (2) the AM Modulator, and (3) the FM Modulator. This block gradually increases the signal level from zero to maximum, along a raised-cosine curve, when one of these blocks is active and the TXA channel state transitions from OFF to ON.

In the case of SSB or DSB, the RF output is proportional to the audio input and there is no RF output until audio is present inside and processed through the TXA unit. The audio input to TXA must pass through the channel's initial up-slew. Hence, the RF output will in-turn up-slew also. However, with the three special cases mentioned above, signal is created internal to the TXA unit independently of audio being present. This signal requires an up-slew to avoid an abrupt transition, consequent "pop" in the audio, and consequent disturbance in the frequency spectrum. In the case of the signal generator, the signal referenced is whatever signal the generator is set to produce. In the case of the AM Modulator and FM Modulator, it is the carrier they generate whose up-slew must be controlled.

There is a control for the length of this Up-Slew. For AM and FM modes, it should be set to a value LESS THAN the Up-Slew specified for the TXA channel – this is so that the carrier will rise faster than the modulating audio.

**void SetTXAuSlewTime (int channel, double time)**

**channel:** identifier/number of the channel being accessed

**time:** length of the up-slew (seconds)

## Siphon

In the block diagram for TXA, after the Siphon block, there are two blocks that can alter the time-domain and frequency-domain content of the signal: (1) PureSignal Predistortion, which, by definition, is intended to predistort the signal to compensate for later non-linearities in the processing chain, and (2) the CFIR which intentionally distorts the frequency response to compensate for frequency-response distortion in later CIC filters in FPGA firmware. It is often desirable to monitor the content of the signal BEFORE these intentional distortions are applied. The Siphon provides the data output for this monitoring. It "siphons" copies of the data from the TXA processing pipeline as the data flows through and either (1) sends the data directly to a WDSP display unit, or (2) makes it available to the console through calls that the console can make to the siphon.

NOTE: Per the setting of an internal parameter, the number of contiguous samples being internally buffered is 4096. Therefore, retrievals of data are limited to this size.

### `void TXASetSipMode (int channel, int mode)`

**channel:** number of the channel being accessed

**mode:** 0 = data is to be retained awaiting a retrieval call from the console; 1 = data is to be sent directly to a specified display unit.

[default = 0]

### `void TXASetSipDisplay (int channel, int disp)`

**channel:** number of the channel being accessed

**disp:** identifier of the display unit to which data is to be sent.

[default = 0]

There are currently three calls the console can make to retrieve data. For historical reasons, at the time of this writing, all return 'float' data. At some future time, calls will likely be added to provide data in the 'double' type.

The following function returns a block of consecutive raw sample values (I-values only) in an array.

### `void TXAGetaSipF (int channel, float* out, int size)`

**channel:** number of the channel being accessed

**out:** pointer to the array where the values are to be stored

**size:** number of values to be returned

The following function returns a block of consecutive raw sample values (complex interleaved I-Q samples) in an array.

### `void TXAGetaSipF1 (int channel, float* out, int size)`

**channel**: number of the channel being accessed

**out**: pointer to the array where the interleaved I-Q values are to be stored

**size**: number of I-Q samples to be returned

The following call returns a set of values of the frequency-domain magnitudes to be used for a frequency-domain display. Values are in dBV, i.e., a sample magnitude of 1.0 yields an output value of 0.0 dB. Per the setting of an internal parameter, **fftsize**, 4096 magnitude values are returned; hence, there is no 'size' parameter.

**void TXAGetaSpecF1 (int channel, float\* out)**

**channel**: number of the channel being accessed

**out**: pointer to the array where the magnitude values are to be stored

There are two possible orderings in which the values from **TXAGetaSpecF1(...)** can be returned. The choice of which to return is determined by the setting of the internal variable **specmode** which is set by the following call.

**void TXASetSipSpecmode (int channel, int mode)**

**channel**: number of the channel being accessed

**mode**: 0 => the "Normal Order" of the FFTW<sup>3</sup> library output is re-arranged such that the two halves of the spectrum are swapped; 1 => the "Normal Order" of the FFTW<sup>4</sup> library output is re-arranged to mirror each half of the FFTW output in place.

[default = 1]

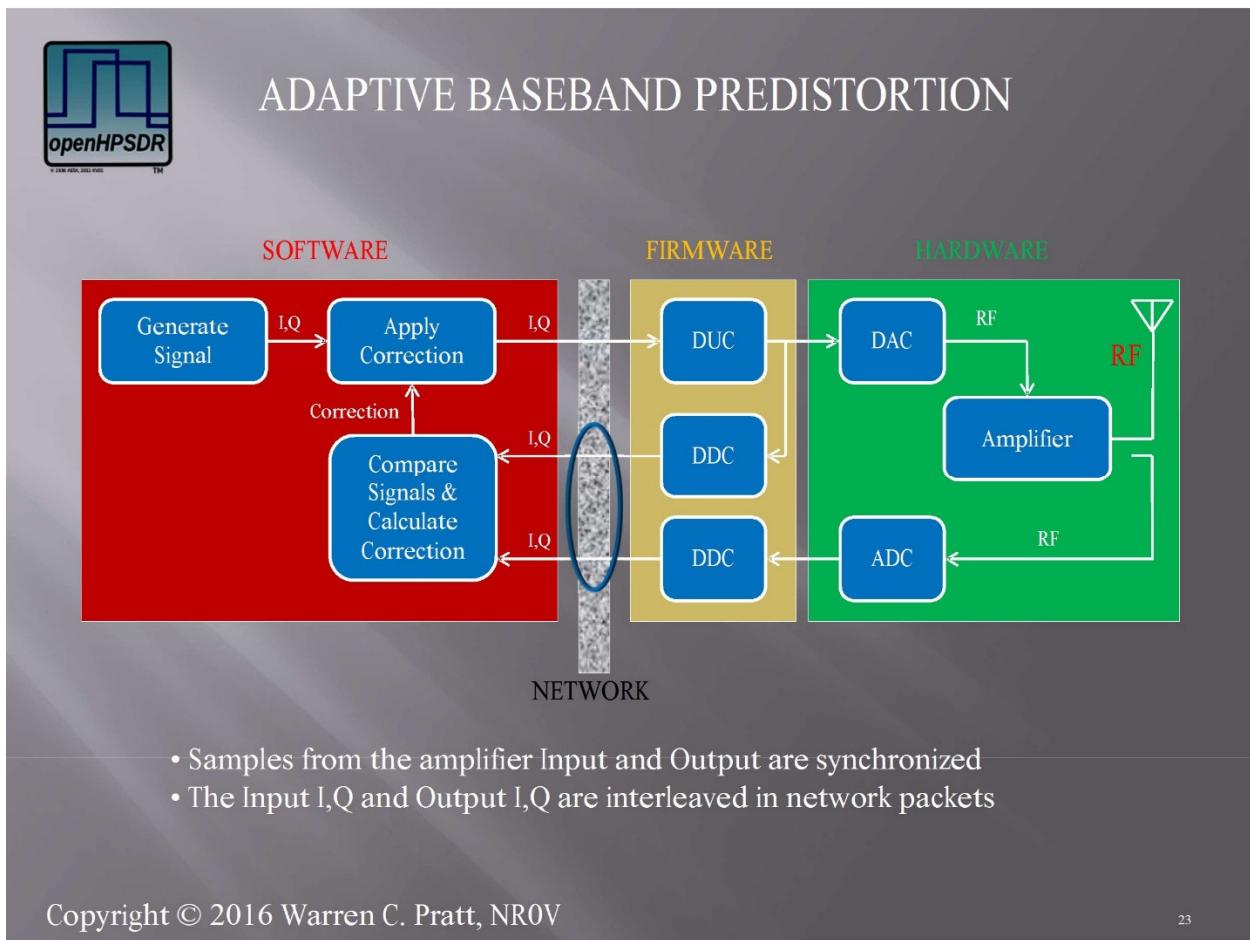
---

<sup>3</sup> FFTW is the FFT library used for WDSP. See <[www.fftw.org](http://www.fftw.org)> for more information.

<sup>4</sup> FFTW is the FFT library used for WDSP. See <[www.fftw.org](http://www.fftw.org)> for more information.

## PureSignal I/Q Predistortion

The following figure displays the operation of PureSignal in the complete SDR environment including software, FPGA firmware, and hardware. The software block "Apply Correction" is where the I-Q signal is modified, applying amplitude and phase correction. The block "Compare Signals & Calculate Correction" executes asynchronously with the dataflow through the TXA unit and uses as its input synchronized samples from the amplifier's input and the amplifier's output. The fact that two sample streams (amplifier input and amplifier output) are required, and the fact that they must be synchronous (each output sample must correspond to the input sample that drove the amplifier to that output), means that some mechanism other than normal channel input and dataflow must be used to provide these samples since the channel structure only supports input of a single sample stream.



### Data Input

Buffers of the two sample streams are provided as input data using the following call.

```
void pscc (int channel, int size, double* tx, double* rx)
```

**channel:** number of the TXA channel being accessed

**size:** number of I-Q samples provided for each sample stream

**tx**: pointer to buffer containing the output-bound sample stream, i.e., the sample stream to drive the amplifier input

**rx**: pointer to buffer containing the sample stream received from the amplifier's output

### **Control**

Considerable information about the state of affairs in PureSignal can be obtained using the following call. This information is often useful in software/firmware debugging and in understanding issues with specific hardware configurations. Some of the information is also used for normal monitoring, for example monitoring of the amplifier output feedback level.

**void GetPSInfo (int channel, int\* info)**

**channel**: number of the channel being accessed

**info**: pointer to an array to hold 16 integer values. Assignment of these 16 values is as follows:

- 0 - status of correction curve builder for rx\_scale
- 1 - status of correction curve builder for cm
- 2 - status of correction curve builder for cc
- 3 - status of correction curve builder for cs
- 4 - amplifier output feedback level
- 5 - count of attempted calibrations
- 6 - results from checking the computed correction solution
- 7 - results from checking the computation of peak power
- ..... UNUSED .....
- 13 - dogcount
- 14 - indicates the state-machine says that iqRun = 1, i.e., the state-machine says that correction values are being applied
- 15 - state of the control state-machine

Data collection and state machine operation is turned OFF/ON by the following call. Nothing happens unless **run** is set to 1. Note that the default is 1 and there are other ways to turn OFF/ON PureSignal operation; hence, this call may be unnecessary in your application.

**void SetPSRunCal (int channel, int run)**

**channel**: number of the channel being accessed

**run**: 0 = OFF; 1 = ON  
[default = 1]

The sample-rate of the two streams provided to **pscc(...)** must be set using the following call.

**void SetPSFeedbackRate (int channel, int rate)**

**channel**: number of the channel being accessed

**rate**: the sample rate (samples / second)

The primary mode of operation is set by the following call. This call provides four control variables that affect the progression of the control state-machine. Note that often the call is made with only one of the last four parameters as '1' and the other three as zero. An example of where multiple parameters may be set to '1' would be in the event that Automatic Calibration Mode is active but we want to change to Single (Manual) Calibration mode. In that case, we can make the call **SetPSControl(channel, 1, 1, 0, 0);**. This call will reset PureSignal, turning off correction. It will also specify the Single/Manual Calibration mode and turn OFF the Automatic Calibration Mode.

#### **void SetPSControl (int channel, int reset, int mancal, int automode, int turnon)**

**channel**: number of the channel being accessed

**reset**: 0 = PS is free to operate as directed; 1 = turn-off PS and reset the control state-machine. To turn OFF PureSignal calibration and correction, call with parameters (1, 0, 0, 0).

**mancal**: Call with parameters (0, 1, 0, 0) to perform a single calibration run and then turn-on correction using that calibration. In this mode, the algorithm is not "adaptive"; it just continues to use this same correction.

**automode**: Call with parameters (0, 0, 1, 0) to turn on auto-calibration mode and correction. In this mode, the algorithm continues looping and re-calibrating, i.e., it is adaptive.

**turnon**: Call with parameters (0, 0, 0, 1) to tell the internal state-machine to continue indefinitely to use whatever correction has been loaded into the correction buffers. This variable is set in situations such as restoring calibration information from a file.

Calibration values can be saved in files and then restored to active correction. These calls save the current calibration and then restore that to active correction.

#### **void PSSaveCorr (int channel, char\* filename)**

**channel**: number of the channel being accessed

**filename**: the file to which the calibration is to be stored

#### **void PSRestoreCorr (int channel, char\* filename)**

**channel**: number of the channel being accessed

**filename**: the file from which the calibration is to be restored

When in automatic-calibration mode, it is possible to add some delay in the re-calibration (adaptation) loop. The benefit would be to slightly reduce CPU utilization. Such delay can be added using the following call.

#### **void SetPSLoopDelay (int channel, double delay)**

**channel**: number of the channel being accessed

**delay**: delay to insert after each calibration (seconds)  
[default = 0.0]

Valid data for calibration can only be collected during transmit, i.e., when MOX is active. The following call conveys the state of MOX to PureSignal.

**NOTE:** SetPSMox should set ‘mox = 0’ BEFORE setting the TXA Channel State to zero/OFF. This will allow certain internal operations to complete during the slew-down of the TXA channel.

#### **void SetPSMox (int channel, int mox)**

**channel**: number of the channel being accessed

**mox**: 0 = not transmitting; 1 = transmitting

After MOX goes true, there is some delay required before valid samples are flowing back to the calibration functions. This delay can be adjusted, if desired, using the following call.

#### **void SetPSMoxDelay (int channel, double delay)**

**channel**: number of the channel being accessed

**delay**: delay to insert before beginning a calibration and after MOX is detected as being active (seconds)  
[default = 0.1]

There is some delay between the two sample streams that must be compared to compute the calibration. This delay is partially due to time delay through the DAC and ADC; however, most of it is due to delay through the analog amplifier circuitry. This delay must be compensated in software to achieve optimum correction. For a solid-state amplifier, it will generally be in the range of 100ns to 1us and will depend upon the band of operation. The desired value of software delay compensation must be provided to the PureSignal software. That is done via the following call.

#### **double SetPSTXDelay (int channel, double delay)**

**return value**: the actual delay the PureSignal provides. The *exact* delay requested may not be available; so, the actual delay provided is returned for comparison.

**channel**: number of the channel being accessed

**delay**: desired delay in seconds

The peak magnitude value of the *returned* outbound sample stream (samples returned in the **tx** buffer for the **pssc(...)** call), i.e., the value when the samples have a value of 1.0 when leaving the TXA software, is a key parameter in computing the correction solution. This value will depend upon scaling in the DUC and DDC firmware/software. The following are calls to "set" this value, meaning to tell the PureSignal software what it is, and to "get" this value which is useful when the console needs to know the default value saved in TXA.

**void SetPSHWPack (int channel, double peak)**

**channel**: number of the channel being accessed

**peak**: peak magnitude value

[default = 0.4072] Note: This is the value for the openHPSDR "old protocol" firmware.

**void GetPSHWPack (int channel, double\* peak)**

**channel**: number of the channel being accessed

**peak**: pointer to location to store the peak magnitude value

When testing new DUC/DDC firmware or software, it is also useful to be able to detect the measured peak magnitude. That is done using the following call. You may wish to call this repeatedly and display the result as it will increase until the maximum is found.

**void GetPSMaxTX (int channel, double\* maxtx)**

**channel**: number of the channel being accessed

**maxtx**: pointer to location to store the maximum measured peak magnitude value

Certain amplifiers have a severe and slow memory effect when their power supply has very poor load regulation. In such cases, relaxing certain PureSignal calibration tolerances may allow calibration to proceed in situations where it otherwise will refuse to do so. A setting is provided for this purpose.

**void SetPSPtol (int channel, double ptol)**

**channel**: number of the channel being accessed

**ptol**: tolerance value. For normal amplifiers, a value of 0.8 is recommended. For "difficult" amplifiers, a value of 0.4 is recommended.

[default = 0.8]

Data points can be retrieved from PureSignal to produce a real-time display of the amplifier gain and phase curves (versus signal magnitude) and of the gain and phase correction curves that are being

applied. This is done for the AmpView display available in the Thetis console application. This is accomplished with the following call.

```
void GetPSDisp (int channel, double* x, double* ym, double* yc, double* ys, double* cm, double* cc,  
double* cs)
```

**channel**: number of the channel being accessed

**x**: pointer to vector for magnitude value of the driving signal (0.0 to 1.0 range). These points are used as reference points to plot **ym**, **yc**, and **ys** data.

**ym**: pointer to vector for amplifier output magnitude

**yc**: pointer to vector for amplifier output cos (phase)

**ys**: pointer to vector for amplifier output sin (phase)

**cm**: pointer to vector for magnitude correction curve values

**cc**: pointer to vector for cos (phase) correction curve values

**cs**: pointer to vector for sin (phase) correction curve values

For each of **x**, **ym**, **yc**, and **ys**, 4096 values are returned.

For **cm**, **cc**, and **cs**, the nested-evaluation coefficients of 16-segment cubic splines are returned (64 values for each of **cm**, **cc**, and **cs**).

Pin-mode applies *a priori* knowledge of the required amplifier gain and phase correction in situations where the collected samples are insufficient for optimal calibration. This need can arise when the radio is not being properly driven to allow sample collection over the entire operating range, or due to some other fault or incorrect operation. This is very useful when there is a significant voltage regulation problem in an amplifier and, therefore, the gain and phase of the amplifier appear somewhat "unstable" as PureSignal collects feedback for calibration.

```
void SetPSPinMode (int channel, int pin)
```

**channel**: number of the channel being accessed

**pin**: 1 turns pin-mode ON; 0 turns pin-mode OFF  
[default = 1]

Map-mode changes the sample-collection requirements to allow easier calibration in situations where the amplifier is detected to be in heavy gain compression. This is done by mapping the collection intervals to a different set of intervals based upon the level of compression. Note that this relaxation in sample-collection requirements MIGHT cause some degradation in high-order IMD. This is adaptive; the

extent of the remapping depends upon the level of compression. If there is no compression, this function has no effect.

#### **void SetPSMapMode (int channel, int map)**

**channel**: number of the channel being accessed

**map**: 1 turns map-mode ON; 0 turns map-mode OFF  
[default = 1]

Stbl-mode substantially separates the "static non-linearity" of the amplifier from the memory effects before computing the correction. This is a step toward further work on memory effects in a future release.

#### **void SetPSSstabilize (int channel, int stbl)**

**channel**: number of the channel being accessed

**stbl**: 1 turns stbl-mode ON; 0 turns stbl-mode OFF  
[default = 1]

Setting '**ints**' and '**spi**' to other than default values can be used to optimize performance. Since the code is currently set up to make all intervals of equal size, this can also be used to implement the "TINT" feature. "TINT" is a largely experimental feature allowing selection of the size of the upper interval within which samples are required to be collected. The dB value reflects the reduction from full-scale. For example, 2.5dB means ~75% of full-scale voltage or ~56% of full-scale power. This should normally be left at the default setting of 0.5dB (~94% of full-scale voltage and ~88% of full-scale power). This feature can be useful in situations where an external processing system, for example a Digital Audio Workstation, is used to precisely control the audio drive amplitude. Note, however, that Auto-Attenuate should probably be disabled when not driving to full-scale as there may not be enough information to accurately calculate its setting which can result in instability in the attenuator value.

#### **void SetPSIntsAndSpi (int channel, int ints, int spi)**

**channel**: number of the channel being accessed

**ints**: number of spline segments to use for the correction curves  
[default = 16]

**spi**: number of samples to collect per spline segment  
[default = 256]

## CFIR Filter

A Compensating FIR (CFIR) filter is generally used in SDR situations where Cascaded Integrator Comb (CIC) filters are used in the Digital Up-Conversion (DUC) or Digital Down-Conversion (DDC) process. CIC filters are commonly used when the DUC or DDC is done in an FPGA as they require very few gates to implement. The CFIR is needed because the CIC filter has a downside that it does not have a flat passband. The CFIR offsets the "droop" in the CIC passband producing a flat frequency response for the combination of the CIC and CFIR. The CFIR can be implemented in either the FPGA or in software. A software implementation is provided here.

Note: For the openHPSDR "USB Protocol"/"Old Protocol", this software function is not used in TXA as it is performed in the FPGA firmware. However, for the openHPSDR "New Protocol", it is used in TXA.

NOTE: At this point, the only call provided for this CFIR in TXA is to turn it OFF/ON. However, there are a number of internal parameters used to specify the shape and characteristics of the filter. These parameters can be made externally available if we have different compensation needs in the future. The internal parameters are currently set to compensate for the CIC filters in the FPGA firmware of the openHPSDR "New Protocol". Key parameters are as follows:

- Input Sample Rate to this filter: `dsp_rate` (channel parameter)
- CIC Input Sample Rate: `output_samplerate` (channel parameter)
- CIC Differential Delay: 1
- CIC Interpolation Factor: 640
- CIC Integrator-Comb Pairs: 5

The following call is used to turn the CFIR OFF/ON.

**void SetTXACFIRRun (int channel, int run)**

`channel`: number of the channel being accessed

`run`: 0 = OFF; 1 = ON

[default = 0]

## TXA Collectives & General Controls

The following call sets the transmission MODE for TXA. This single call turns OFF/ON and provides certain required settings to blocks such as the AM and FM modulators.

### **void SetTXAMode (int channel, int mode)**

**channel:** number of the channel being accessed

**mode:** 0 = LSB; 1 = USB; 2 = DSB; 5 = FM; 6 = AM; 7 = DIGU; 8 = SPEC; 9 = DIGL; 12 = AM\_LSB; 13 = AM\_USB. [default = 0]

The following collective is used to set the length of filter impulse responses within TXA. This is the **PREFERRED** method, compared to setting the filters individually. Note that when these filters were created, their 'nc' was set to "max (2048, dsp\_size)". A minimum of 2048 is recommended; although, 1024 may be acceptable in some cases.

### **void TXASetNC (int channel, int nc)**

**channel:** number of the channel being accessed

**nc:** length of filter impulse responses. **This must be a power-of-two and must be greater than or equal to the `dsp_size` currently in use.**

The following collective is used to choose Linear Phase versus Low Latency for filters within TXA. This is the **PREFERRED** method, compared to setting the filters individually.

### **void TXASetMP (int channel, int mp)**

**channel:** number of the channel being accessed

**mp:** 0 = Linear Phase; 1 = Low Latency (Minimum Phase).

## Panadapter & Other Frequency-Domain Displays

WDSP contains functionality to accept raw wide-bandwidth time-domain samples and, from them, repetitively create the data ("pixel values") for panadapter displays and other frequency-domain displays. This full-featured implementation includes support for:

- combining or interpolating FFT output bins, as needed, to create a requested number of pixel values across the screen,
- symmetrical and asymmetrical clipping to support pan and zoom functionality,
- multiple averaging modes,
- multiple detectors (peak, average, rosenfell, and sample),
- normalization to a one Hertz bandwidth for noise measurements,
- multiple pixel outputs for the same input data set, for example, the panadapter and waterfall can be using different averaging modes,
- software spur elimination for spectrum analyzer applications,
- stitched display for viewing a spectrum exceeding the bandwidth of a single sample stream,
- both real and complex input samples,
- a selection of window functions used to window the input data, and
- curve-fitted amplitude calibration facilities.

Instances of such "Displays" are not created within RXA or TXA. These Displays must be separately instantiated. A typical scenario would be to create one Display for each RXA and for each TXA channel and also perhaps one Display for each ADC such that a view can simultaneously be provided for the entire frequency spectrum digitized by the ADC. However, if computing resources are constrained, it is also possible to create less Displays and share them among multiple views. For example, a single Display could be shared between being a panadapter during receive and displaying the frequency spectrum of the out-going transmitted signal during transmit.

The display code was originally written for a different project with different objectives and was then later included within WDSP. Therefore, there are some style differences between this Display code and the rest of WDSP. Expect that this code is likely to become more similar to the rest of WDSP over time, as development time permits.

The Display unit receives calls to set various parameters, to provide it with input data, and to retrieve "pixel values" from it. These calls to a Display unit may be made from either within an RXA or TXA channel or from outside WDSP entirely, depending upon the needs. For example, one of the supported features of the Display is to "stitch" multiple sub-spectra together to create a view of a wider received spectrum. However, a WDSP channel only supports the receipt of one continuous sample stream. Since stitching requires multiple continuous sample streams, calls to supply the input data to the Display would generally be made outside of the WDSP channel. On the other hand, for a view of the Spectrum data after the bandpass filter, the call to supply input data to the Display would need to be made within the WDSP channel.

## Creating and Destroying a Display

Creating a display consists of allocating all the resources it requires. These resources are a function of things like the maximum FFT size to be used, the maximum number of LO positions to be used for spur elimination (a spectrum analyzer feature), and the maximum number of sub-spans to be stitched together to form the complete spectrum. A Display is created using the call:

```
void XCreateAnalyzer (int disp, int* success, int m_size, int m_LO, int m_stitch, char* app_data_path)
```

**disp**: number/identifier of the Display being created. This value currently ranges from 0 through 63. If one display is to be created per DSP channel, it may prove convenient to have the 'disp' value match the channel identifier.

**success**: a pointer to a return value that will be zero if the display has been successfully created.

**m\_size**: the maximum FFT size to be used. The WDSP Wisdom calculations must include any selected values and currently calculate power-of-two values through 262144. This value must be a power-of-two.

**m\_LO**: the maximum number of Local Oscillator positions to be used for spur elimination. This is a spectrum analyzer feature that provides comparison of spectra produced from multiple LO frequencies to automatically eliminate spurious responses. For non-spectrum-analyzer use, set this to 1.

**m\_stitch**: the maximum number of sub-spans that will be stitched together to create a single spectrum to be displayed. For example, if no more than three firmware receiver sample streams are to be used to create a single display, set this to 3.

**app\_data\_path**: not used.

When the display is no longer needed, for example when the host application is preparing to close, its resources are freed using this call:

```
void DestroyAnalyzer (int disp)
```

**disp**: the identifier of the Display whose resources are to be freed.

## Setting Display Parameters

The `SetAnalyzer(...)` call is used to set many of the parameters for the display unit.

```
void SetAnalyzer (int disp, int n_pixout, int n_fft, int typ, int* flp, int sz, int bf_sz, int win_type, double pi, int ovrlp, int clp, double fscLin, double fscHin, int n_pix, int n_stch, int calset, double fmin, double fmax, int max_w)
```

`disp`: the identifier of the Display whose parameters are to be set.

`n_pixout`: the number of pixel outputs being used. For example, this could be 1 if the same pixel-value data is to be used for the panadapter and waterfall; or, it could be 2 if different detectors or averaging modes are desired for the panadapter and waterfall.

`n_fft`: the number of LO positions to be used for spur elimination (a spectrum analyzer feature). This should be set to 1 if spur elimination is not being used.

`typ`: specifies whether REAL or COMPLEX sample data is to be provided. 0 for REAL which must be delivered in the 'I' channel; 1 for COMPLEX which will be delivered in I and Q channels.

`flp`: pointer to a vector with one element per LO frequency. The elements are used to specify whether the LO frequencies are on the low side or high side of the operating range. For non-SA use, this will be a single element with a value of 0. For example:

```
int[] flp = { 0 };
```

`sz`: size of the FFT to be used. This must be a power of two.

`bf_sz`: number of samples to be provided each time a call to provide input samples is made.

`win_type`: is used to specify the type of window function to be used for processing. Beginning with 0, assignments are: Rectangular, 4-term Blackman-Harris, Hann, Flat-top, Hamming, Kaiser, 7-term Blackman-Harris. For general purpose use, 5 (Kaiser) and 6 (7-term Blackman-Harris) are good choices. For best amplitude accuracy, use 3 (Flat-top).

`pi`: parameter used to shape the Kaiser window function. 14.0 is a reasonable default value.

`ovrlp`: the number of samples to be re-used from the previous FFT. For SA use, overlap can reveal transient signals that may otherwise not be displayed. For general purpose radio use, overlap allows the display frame-rate to be independent of the sample-rate. For example, at an FFT size of 262144 and a sample rate of 192000, it takes over one second to capture enough samples for an entirely new display frame. However, by re-using some number of samples from the previous FFT, we can create new frames at a much faster rate and have an interactive display. Overlap can be calculated as:

```
//set overlap as needed to achieve the desired frame_rate
overlap = (int)Math.Max(0.0, Math.Ceiling(fft_size - (double)sample_rate /
(double)frame_rate));
```

where 'frame\_rate' is in frames/second.

**clp**: the number of FFT bins to be (symmetrically) clipped from each side of each sub-span. Preceding firmware decimation filters (in either FPGA firmware or Direct Fourier Conversion code) are not perfectly rectangular --- they have a roll-off on each side of the spectrum. It is generally not desirable to display the roll-off area, especially if stitching is to be used. A primary use of this capability is to clip off those bins. The value for 'clp' can generally be set by specifying some fraction of the spectrum to be clipped on each side. For example:

```
//fraction of the spectrum to clip off each side of each sub-span
const double CLIP_FRACTION = 0.017;
//clip is the number of bins to clip off each side of each sub-span
clp = (int)Math.Floor(CLIP_FRACTION * fft_size);
```

**fscLin**: the number of bins to clip from low side of the entire (stitched) span. This parameter can be used along with **fscHin** to PAN and ZOOM within the spectrum represented by the entire stitched span. Based upon PAN and ZOOM settings, these parameters can be set as follows:

```
//the amount of frequency in each fft bin is given by:
//  this is also equal to the interval width!
double bin_width = (double)sample_rate / (double)fft_size;

//the number of useable bins per subspan is
//  the '-1' is due to clipping the Nyquist bin
int bins_per_subspan = fft_size - 1 - 2 * clip;

//the amount of useable bandwidth we get from each subspan is:
//  we'd subtract '1' from 'bins_per_subspan' if we wanted the
//  interval_width_per_subspan
bw_per_subspan = bins_per_subspan * bin_width;

//the total number of bins available to display is:
int bins = stitches * bins_per_subspan;

//the number of intervals among all the bins equals 'bins - 1'
double intervals = (double)(bins - 1);

//apply log function to zoom slider value
double zoom_slider = Math.Log10(9.0 * z_slider + 1.0);

//limits how much you can zoom in; higher value means you zoom more
const double zoom_limit = 100;

//calculate the width in intervals after applying zoom
double width = intervals * (1.0 - (1.0 - 1.0 / zoom_limit) * zoom_slider);

//fscLin is 0 if pan_slider is 0; it's 'intervals - width' if
//pan_slider is 1
//fscHin is 'intervals - width' if pan_slider is 0; it's 0 if
//pan_slider is 1
double fscLin = pan_slider * (intervals - width);
double fscHin = intervals - width - span_clip_l;
```

**fscHin:** See above comments about fsclIn.

**n\_pix:** number of pixel values to be retrieved upon each call to retrieve pixels, i.e., the width, in pixels, of the window in which the results are to be displayed.

**n\_stch:** number of sub-spans to be stitched. If only one DDC is to be used, this is 1. If, for example, the sample streams from 3 DDCs are to be stitched, this is 3.

**calset:** specifies which of multiple sets of calibration data is to be used to adjust signal amplitude across the range of the spectrum. If the calibration feature is not in use, set this value to 0.

**fmin:** the minimum frequency of the span (used for calibration). If calibration is not in use, set this value to 0.0.

**fmax:** the maximum frequency of the span (used for calibration). If calibration is not in use, set this value to 0.0.

**max\_w:** a parameter used to determine how much data will be held in display buffers. It is calculated as:

```
const double KEEP_TIME = 0.1;
max_w = fft_size + (int)Math.Min(KEEP_TIME * sample_rate, KEEP_TIME *
    fft_size * frame_rate);
```

When the calibration feature is used, the following call is to set the calibration data. A table is to be provided with two columns, (0) frequency and (1) the calibration multipliers corresponding to those frequencies. The 'calibration multipliers' are NOT dB values to be added, they are linear multipliers. This table is interpolated using a cubic spline to create a smooth calibration curve.

### **void SetCalibration (int disp, int set\_num, int n\_points, double(\* cal)[2])**

**disp:** the identifier of the Display whose calibration is being set.

**set\_num:** the identifier for this calibration data set.

**n\_points:** the number of calibration points in the set; i.e., the number of rows in the following table.

**cal:** pointer to the table of calibration data.

The following call is used to select the detector-type for a particular Display and Pixel Output from that Display. The positive-peak detector is the best selection for measuring the amplitude of a signal. The average and sample detectors are provided specifically for noise measurements and the output is

adjusted for the chosen window's equivalent noise bandwidth when using these two detectors. The rosenfell detector is common in spectrum analyzers and gives an indication of both signal and noise.

**void SetDisplayDetectorMode (int disp, int pixout, int mode)**

**disp**: identifier for the Display.

**pixout**: identifier of the pixel output for which the detector is being set.

**mode**: detector mode. 0 = positive peak; 1 = rosenfell; 2 = average; 3 = sample; 4 = rms (across fft bins within the pixel).

The following call is used to select the averaging-mode for a particular Display and Pixel Output from that Display.

**void SetDisplayAverageMode (int disp, int pixout, int mode)**

**disp**: identifier for the Display.

**pixout**: identifier of the pixel output for which the averaging mode is being set.

**mode**: averaging mode. -1 = peak-hold; 0 = no averaging; 1 = weighted averaging of linear data; 2 = time-window averaging of linear data; 3 = weighted averaging of log data. This last mode, weighted averaging of log data, looks nice on the panadapter log scale; however, it is not mathematically accurate for time-varying signals or noise measurements.

When time-window averaging of data is used, the number of display frames to average needs to be specified. The following call performs that operation.

**void SetDisplayNumAverage (int disp, int pixout, int num)**

**disp**: identifier for the Display.

**pixout**: identifier of the pixel output for which the value is being set.

**num**: number of frames to average.

When weighted averaging is used, a multiplier value must be supplied. Averaging then proceeds as:  
new\_display\_value = mult \* old\_display\_value + (1.0 - mult) \* new\_pixel\_value;

**void SetDisplayAvBackmult (int disp, int pixout, double mult)**

**disp**: identifier for the Display.

**pixout**: identifier of the pixel output for which the value is being set.

**mult**: multiplier value.

The display code must know the sample-rate of the input data to be able to normalize measurements to a 1.0 Hz bandwidth, when desired. The bandwidth is set by:

**void SetDisplaySampleRate (int disp, int rate)**

**disp**: identifier for the Display.

**rate**: sample-rate (samples per second).

Pixel values will be normalized to a 1.0 Hz bandwidth if '**norm**' is set to 1.

**void SetDisplayNormOneHz (int disp, int pixout, int norm)**

**disp**: identifier for the Display.

**pixout**: identifier of the pixel output for which the parameter is being set.

**norm**: 0 = do not normalize; 1 = normalize to one Hz bandwidth.

## Supplying Input Data

Several input functions have been developed over time to meet varying console needs.

When I and Q are kept in separate buffers, as opposed to being interleaved in a single buffer, use the `Spectrum(...)` call. Note that the type of the I/Q data is specified as `dINREAL` which is defined as either `double` or `float` in the file "comm.h".

```
void Spectrum (int disp, int ss, int LO, dINREAL* pI, dINREAL* pQ)
```

`disp`: identifier for the Display.

`ss`: sub-span (of a stitched display) for which data is being supplied. If stitching is not being used, `ss` = 0.

`LO`: LO position for which data is being supplied. (for Spectrum Analyzer software spur removal)  
If spur removal is not being used, `LO` = 0.

`pI`: pointer to the array of I values

`pQ`: pointer to array of Q values

When I and Q are interleaved in a single buffer, one generally uses the `Spectrum2(...)` call. Note that the type of the I/Q data is specified as `dINREAL` which is defined as either `double` or `float` in the file "comm.h".

```
void Spectrum2 (int run, int disp, int ss, int LO, dINREAL* pbuff)
```

`run`: 0 = ignore input data and return; 1 = copy input data from `pbuff` and process.

`disp`: identifier for the Display.

`ss`: sub-span (of a stitched display) for which data is being supplied. If stitching is not being used, `ss` = 0.

`LO`: LO position for which data is being supplied. (for Spectrum Analyzer software spur removal)  
If spur removal is not being used, `LO` = 0.

`pbuff`: pointer to the array of interleaved I/Q values

In the event interleaved I/Q data is to be supplied as `double`, but because float data is at times being supplied from elsewhere `dINREAL` is defined as `float`, the `Spectrum0(...)` call may be used.

```
void Spectrum0 (int run, int disp, int ss, int LO, double* pbuff)
```

`run`: 0 = ignore input data and return; 1 = copy input data from `pbuff` and process.

`disp`: identifier for the Display.

**ss**: sub-span (of a stitched display) for which data is being supplied. If stitching is not being used, ss = 0.

**LO**: LO position for which data is being supplied. (for Spectrum Analyzer software spur removal)  
If spur removal is not being used, LO = 0.

**pbuf**: pointer to the array of interleaved I/Q values

## Retrieving Pixel Values

As previously discussed, new frames of data are calculated periodically based upon several settings. Pixel values should be periodically retrieved, at this same desired frame-rate, using `GetPixels(...)`. If new data is available (indicated by `flag = 1`), it should be drawn. If not, check again later -- nothing new to draw now.

```
void GetPixels (int disp, int pixout, dOUTREAL* pix, int* flag)
```

`disp`: identifier for the Display.

`pixout`: identifier of the pixel output from which pixel values are to be retrieved.

`pix`: pointer to buffer into which the pixel values (dBm) will be placed.

`flag`: pointer to an integer that will be set to 1 if new data was available and will be set to 0 if no new data was available.

## Clearing the Display Pipeline

If it is desired to clear the in-process data from the display pipeline, the following call can be used. This might be used, for example, on a TX/RX transition for a continuously running display.

```
void ResetPixelBuffers (int disp)
```

`disp`: identifier for the Display.

## Retrieving Maximum Bin Value (dB) Within a Frequency Range

A frequency range, relative to the center\_frequency of the DDC driving the display, can be specified and the maximum value of any display FFT bin within that range will be returned, in dB. Note that if adjustments are normally applied to display ‘pixel’ values, those same adjustments should be applied to this value.

This functionality must be declared for each ‘`disp`’ for which the functionality is desired. Call this function for initial setup AND anytime one of the specified parameters changes.

```
void SetupDetectMaxBin (int run, int disp, int ss, int LO, double rate, double fLow, double fHigh, double tau, int frame_rate)
```

`run`: Set to ‘1’ if this feature is in use, ‘0’ otherwise. Note that CPU cycles can be saved by setting this to ‘0’ when not in use.

`disp`: identifier for the Display.

**ss**: sub-span. Set to '0' if stitching is not in use. Note that only one 'ss'/'LO pair is permitted, per display, at a time.

**LO**: LO position. Set to '0' if spur-removal is not in use. Note that only one 'ss'/'LO pair is permitted, per display, at a time.

**rate**: current sample-rate of display data, e.g., 192000.0.

**fLow**: Lowest frequency of the frequency range to evaluate, referenced to the center\_frequency to which the DDC is tuned. For example, for LSB, not using CTUN, a typical value would be '-3000.0'.

**fHigh**: Highest frequency of the frequency range to evaluate, referenced to the center\_frequency to which the DDC is tuned. For example, for LSB, not using CTUN, a typical value would be '-300.0'.

**tau**: Metering decay time-constant, in seconds. Useful when the bin-value is used to drive a meter. Typical value might be '0.5'.

**frame\_rate**: Display frame\_rate current in use, e.g., '60'.

The following function returns the maximum bin value (dB) for the selected display. It may be called at any time and will return the value from the most recent FFT.

**double GetDetectMaxBin (int disp)**

**disp**: identifier for the Display.

## Blocks Used Outside Channels

There are several blocks included with the WDSP library that are often utilized OUTSIDE of the WDSP channels. Instantiating and accessing those blocks is discussed here.

### Preemptive Wideband Noise Blanker

Noise blankers attempt to remove "impulse noise", brief time-domain impulses imposed upon the I-Q sample data. These impulses are often caused by things like power line arcing and may or may not be repetitive in nature. To remove these impulses, noise blankers must do two things: (1) detect the samples that have been corrupted by the impulses, and (2) replace those samples with some estimate of the original signal.

In the case of the Preemptive Wideband Noise Blanker, detection is done by comparing the power contained in each sample with the time-averaged power across the wide bandwidth seen by the blunker. "Wideband" implies that the sample stream has a high enough sample-rate to support the bandwidth of roughly the size of a typical amateur band, for example, a few hundred KHz for HF operation. Processing a wider bandwidth in the blunker has little if any benefit since the input bandwidth is usually restricted anyway by input bandpass filters or by the bandwidth of the antenna itself.

Correction, in this blunker, is done by replacing a sequence of samples believed to be corrupt with an estimate value of zero. However, there are several other considerations.

- We should not transition instantaneously from the uncorrupted signal level to zero and from zero back to the uncorrupted signal level; this would create a disturbance in the frequency spectrum and audible clicks.
- The leading and trailing edges of the impulse may not be detected even though the higher magnitude center is. Therefore, we may wish to "advance" the blanking to before the detected impulse and "hang" onto the blanking slightly after the impulse appears to have passed.

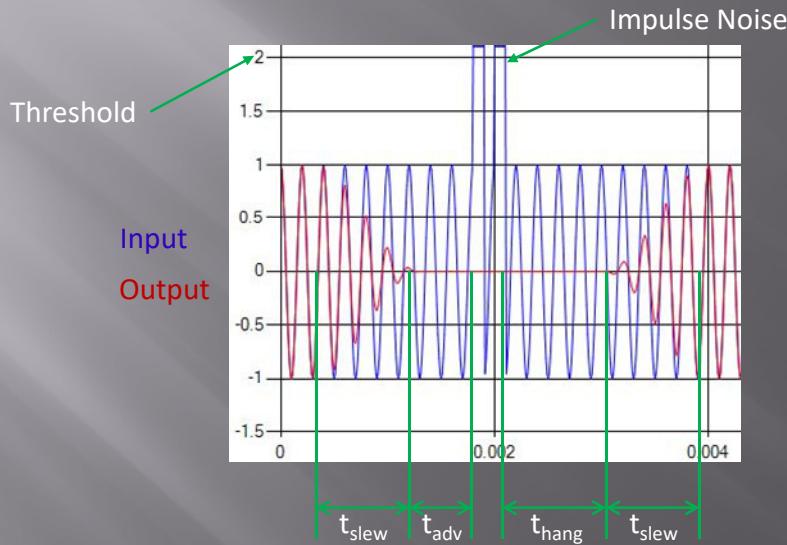
The following figure illustrates the operation of the Preemptive Wideband Noise Blanker. Signal amplitude slewing to zero and back, as well as advance and hang times are illustrated. Note also that the threshold at which a sample is judged to be part of an impulse is adjustable.

It is desirable to have the wideband blankers also affect the panadapter and waterfall displays. Since the data input for these occurs outside an RXA channel, the wideband blunker must be called from outside an RXA channel also.



## PREEMPTIVE WIDEBAND BLANKER “NB”

NR	ANF
NB	SNB
MUT	BIN
MNF	



Copyright © 2016 Warren C. Pratt, NROV

There are two sets of calls that can be used to access the Preemptive Wideband Blanker:

- 1) When coding in C, one may include the "nob.h" header file and use calls that directly reference the pointer to the noise blanker data structure which is of type **ANB**. The 'create' function, creating an instance of the blanker, returns a pointer of type **ANB** and then this pointer is subsequently used in all communications with this instance of the blanker.
- 2) The more general set of calls requires the specification of an integer 'identifier' in the call to the create function. This identifier is then subsequently used in all communications with this same instance of the blanker.

### **Pointer-Based Calls**

A blanker is instantiated (its resources are allocated and default values set) using the following call.

```
ANB create_anb (int run, int bufsize, double* in, double* out, double samplerate, double tau, double hangtime, double advtime, double backtau, double threshold)
```

**run**: default value of the **run** variable. 0 = OFF; 1 = ON. When this block is turned OFF, the input buffer is copied to the output buffer in the event the buffers are not the same.

**bufsize**: default number of complex I-Q samples in the input buffer.

**in**: default pointer to the input buffer.

**out**: default pointer to the output buffer which may be the same as the input buffer.

**samplerate**: default sample-rate.

**tau**: default duration (seconds) of the raised-cosine transitions between normal signal levels and the zero estimate during an impulse,  $t_{slew}$ . 0.0001 is a reasonable starting value, depending upon sample-rate.

**hangtime**: default hang-time,  $t_{hang}$  (seconds). 0.0001 is a reasonable starting value, depending upon sample-rate.

**advtme**: default advance-time,  $t_{adv}$  (seconds). 0.0001 is a reasonable starting value, depending upon sample-rate.

**backtau**: default time-constant for averaging power across the wide bandwidth. 0.05 is a reasonable starting value.

**threshold**: default threshold. A sample is judged to be part of an impulse if its magnitude is greater than **threshold**\***average\_power**. An appropriate range for threshold is approximately 15.0 to 500.0. If **threshold** is too low, normal signals may be wrongly identified as impulse noise. If it is too high, successful detection will not occur.

Resources are freed using the following call.

### **void destroy\_anb (ANB a)**

**a**: pointer to the blunker data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

Buffers are flushed of stale data and indexes are re-initialized using the following call.

### **void flush\_anb (ANB a)**

**a**: pointer to the blunker data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

A buffer of input data is processed and a buffer of output data written using the following call.

### **void xanb (ANB a)**

**a**: pointer to the blunker data structure.

Various parameters are reset using the following calls. Unless otherwise noted, these following calls may be made at any time including while data is flowing.

**void setBuffers\_anb (ANB a, double\* in, double\* out)**

**a**: pointer to the blunker data structure.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer which may be the same as the input buffer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

**void setSamplerate\_anb (ANB a, int rate)**

**a**: pointer to the blunker data structure.

**rate**: sample-rate.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

**void setSize\_anb (ANB a, int size)**

**a**: pointer to the blunker data structure.

**size**: number of complex I-Q samples in the input buffer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

**void pSetRCVRANBRun (ANB a, int run)**

**a**: pointer to the blunker data structure.

**run**: 0 = OFF, 1 = ON. When OFF, the input buffer will be copied to the output buffer if they are not the same.

**void pSetRCVRANBTau (ANB a, double tau)**

**a**: pointer to the blunker data structure.

**tau**: duration (seconds) of the raised-cosine transitions between normal signal levels and the zero estimate during an impulse,  $t_{slew}$ .

**void pSetRCVRANBHangtime (ANB a, double time)**

**a**: pointer to the blunker data structure.

**time**: hang-time,  $t_{hang}$  (seconds).

**void pSetRCVRANBAdvtime (ANB a, double time)**

**a**: pointer to the blunker data structure.

**time**: advance-time,  $t_{adv}$  (seconds).

**void pSetRCVRANBBacktau (ANB a, double tau)**

**a**: pointer to the blunker data structure.

**tau**: time-constant for averaging power across the wide bandwidth.

**void pSetRCVRANBThreshold (ANB a, double thresh)**

**a**: pointer to the blunker data structure.

**thresh**: detection threshold. (See explanation above in description of 'create' call.)

### *Identifier-Based Calls*

A blunker is instantiated (its resources are allocated and default values set) using the following call.

**ANB create\_anbEXT (int id, int run, int bufsize, double samplerate, double tau, double hangtime, double advtime, double backtau, double threshold)**

**id**: an integer value used as the identifier for this instance of the blunker. The number of blankers is currently restricted (by the setting of an internal variable) to 32. Correspondingly, identifiers must be in the range 0 through 31.

**run**: default value of the **run** variable. 0 = OFF; 1 = ON. When this block is turned OFF, the input buffer is copied to the output buffer in the event they are not the same.

**bufsize**: default number of complex I-Q samples in the input buffer.

**samplerate**: default sample-rate.

**tau**: default duration (seconds) of the raised-cosine transitions between normal signal levels and the zero estimate during an impulse,  $t_{slew}$ . 0.0001 is a reasonable starting value, depending upon sample-rate.

**hangtime**: default hang-time,  $t_{hang}$  (seconds). 0.0001 is a reasonable starting value, depending upon sample-rate.

**advtime**: default advance-time,  $t_{adv}$  (seconds). 0.0001 is a reasonable starting value, depending upon sample-rate.

**backtau**: default time-constant for averaging power across the wide bandwidth. 0.05 is a reasonable starting value.

**threshold**: default threshold. A sample is judged to be part of an impulse if its magnitude is greater than **threshold\*average\_power**. An appropriate range for threshold is approximately 15.0 to 500.0. If threshold is too low, normal signals may be wrongly identified as impulse noise. If it is too high, successful detection will not occur.

Resources are freed using the following call.

#### **void destroy\_anbEXT (int id)**

**id**: identifier for this instance of the blanker.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

Buffers are flushed of stale data and indexes are re-initialized using the following call.

#### **void flush\_anbEXT (int id)**

**id**: identifier for this instance of the blanker.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

A buffer of input data is processed and a buffer of output data written using the following call.

#### **void xanbEXT (int id, double\* in, double\* out)**

**id**: identifier for this instance of the blanker.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer which may be the same as the input buffer.

Various parameters are reset using the following calls. Unless otherwise noted, these following calls may be made at any time including while data is flowing.

#### **void SetEXTANBRun (int id, int run)**

**id**: identifier for this instance of the blanker.

**run**: 0 = OFF, 1 = ON. When OFF, the input buffer will be copied to the output buffer if they are not the same.

#### **void SetEXTANBSamplerate (int id, int rate)**

**id**: identifier for this instance of the blanker.

**rate**: sample-rate.

**void SetEXTANBBuffsize (int id, int size)**

**id**: identifier for this instance of the blunker.

**size**: number of complex I-Q samples in the input buffer.

**void SetEXTANBTau (int id, double tau)**

**id**: identifier for this instance of the blunker.

**tau**: duration (seconds) of the raised-cosine transitions between normal signal levels and the zero estimate during an impulse,  $t_{slew}$ .

**void SetEXTANBHangtime (int id, double time)**

**id**: identifier for this instance of the blunker.

**time**: hang-time,  $t_{hang}$  (seconds).

**void SetEXTANBAdvtme (int id, double time)**

**id**: identifier for this instance of the blunker.

**time**: advance-time,  $t_{adv}$  (seconds).

**void SetEXTANBBacktau (int id, double tau)**

**id**: identifier for this instance of the blunker.

**tau**: time-constant for averaging power across the wide bandwidth.

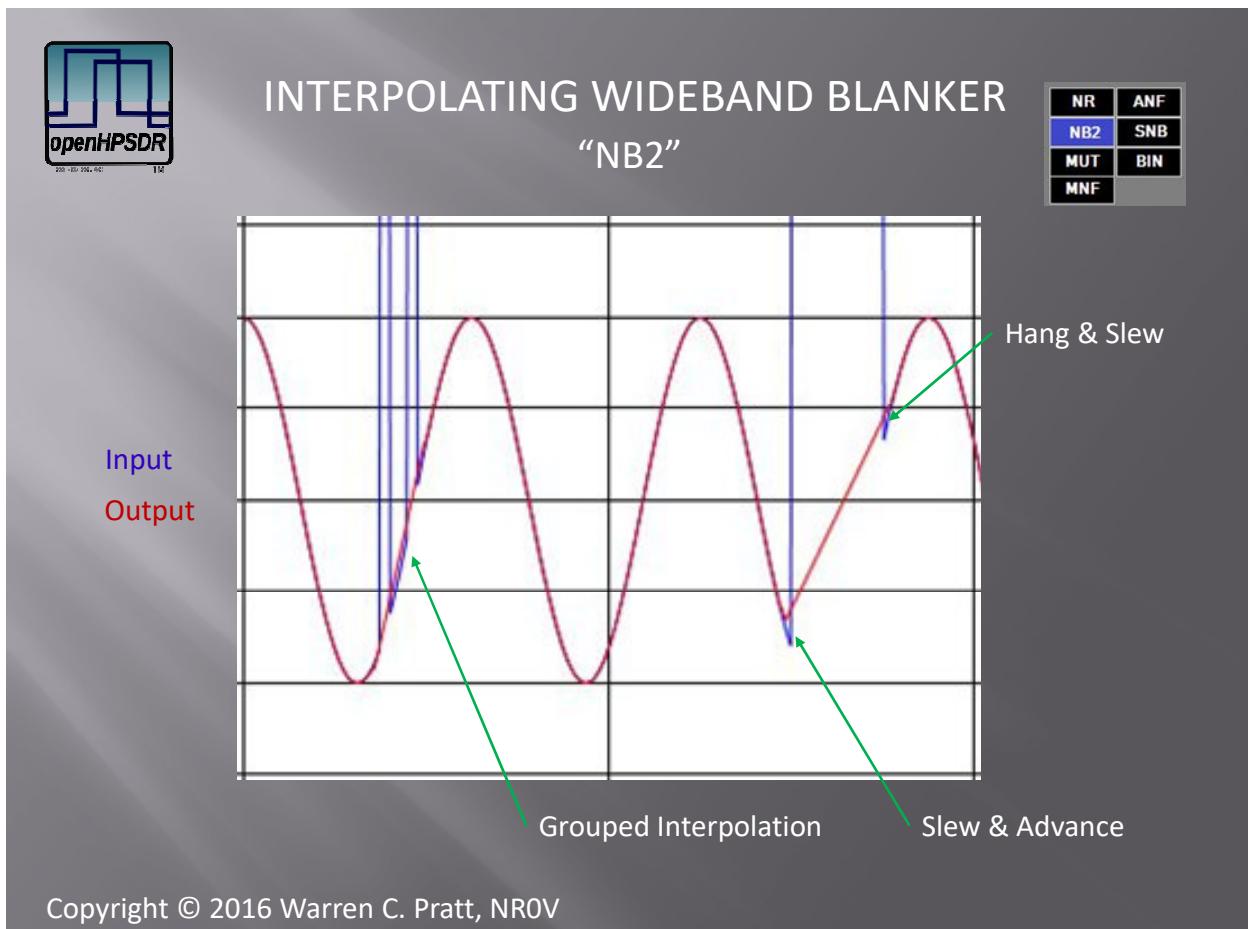
**void SetEXTANBANBThreshold (int id, double thresh)**

**id**: identifier for this instance of the blunker.

**thresh**: detection threshold. (See explanation above in description of 'create' call.)

## Interpolating Wideband Noise Blanker

The Interpolating Wideband Noise Blanker is similar to the Preemptive Wideband Noise Blanker except that it has additional modes to estimate the original signal, rather than just estimating it as zero as is done by the Preemptive Wideband Noise Blanker. As shown in the following figure, it can do things like interpolate across impulses as its estimate of the original signal.



There are two sets of calls that can be used to access the Interpolating Wideband Blanker:

- 1) When coding in C, one may include the "nobll.h" header file and use calls that directly reference the pointer to the noise blunker data structure which is of type **NOB**. The 'create' function, creating an instance of the blunker, returns a pointer of type **NOB** and then this pointer is subsequently used in all communications with this instance of the blunker.
- 2) The more general set of calls requires the specification of an integer 'identifier' in the call to the create function. This identifier is then subsequently used in all communications with this same instance of the blunker.

### Pointer-Based Calls

A blunker is instantiated (its resources are allocated and default values set) using the following call.

**NOB create\_nob (int run, int bufsize, double\* in, double\* out, double samplerate, int mode, double advslewtime, double advtime, double hangslewtime, double hangtime, double max\_imp\_seq\_time, double backtau, double threshold)**

**run**: default value of the **run** variable. 0 = OFF; 1 = ON. When this block is turned OFF, the input buffer is copied to the output buffer in the event they are not the same.

**bufsize**: default number of complex I-Q samples in the input buffer.

**in**: default pointer to the input buffer.

**out**: default pointer to the output buffer which may be the same as the input buffer.

**samplerate**: default sample-rate.

**mode**: default mode. **mode** determines what estimate values are used for the sequence of corrupt samples to be replaced. 0 = zero mode (estimate as zero); 1 = sample-hold (take the value of non-corrupt signal at the beginning of the impulse and hold that throughout the corrupt sequence); 2 = mean-hold (average the non-corrupt values at the beginning and end of the corrupt sequence and use that as the estimate during the corrupt sequence); 3 = hold-sample (take the value of non-corrupt signal at the end of the impulse and hold that throughout the corrupt sequence); 4 = linearly interpolate across the corrupt sequence.

**advslewtime**: default time (seconds) to slew from the non-corrupt signal before the advance time. 0.0001 is a reasonable initial value for typical sample rates.

**advtime**: default advance time (seconds). 0.0001 is a reasonable initial value for typical sample rates.

**hangslewtime**: default time (seconds) to slew after the hang time back to the non-corrupt signal. 0.0001 is a reasonable initial value for typical sample rates.

**hangtime**: default hang time (seconds). 0.0001 is a reasonable initial value for typical sample rates.

**max\_imp\_seq\_time**: default maximum amount of time (seconds) across which a closely spaced sequence of impulses will be evaluated. 0.025 is a reasonably conservative value.

**backtau**: default time-constant for averaging power across the wide bandwidth. 0.05 is a reasonable starting value.

**threshold**: default threshold. A sample is judged to be part of an impulse if its magnitude is greater than **threshold\*average\_power**. An appropriate range for threshold is approximately 15.0 to 500.0. If threshold is too low, normal signals may be wrongly identified as impulse noise. If it is too high, successful detection will not occur.

Resources are freed using the following call.

### **void destroy\_nob (NOB a)**

**a**: pointer to the blanker data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

Buffers are flushed of stale data and indexes are re-initialized using the following call.

### **void flush\_nob (NOB a)**

**a**: pointer to the blanker data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

A buffer of input data is processed and a buffer of output data written using the following call.

### **void xnob (NOB a)**

**a**: pointer to the blanker data structure.

Various parameters are reset using the following calls. Unless otherwise noted, these following calls may be made at any time including while data is flowing.

### **void setBuffers\_nob (NOB a, double\* in, double\* out)**

**a**: pointer to the blanker data structure.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer which may be the same as the input buffer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

### **void setSamplerate\_nob (NOB a, int rate)**

**a**: pointer to the blanker data structure.

**rate**: sample-rate.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

### **void setSize\_nob (NOB a, int size)**

**a**: pointer to the blanker data structure.

**size:** number of complex I-Q samples in the input buffer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void pSetRCVRNOBRun (NOB a, int run)**

**a:** pointer to the blunker data structure.

**run:** 0 = OFF, 1 = ON. When OFF, the input buffer will be copied to the output buffer if they are not the same.

#### **void pSetRCVRNOBMode (NOB a, int mode)**

**a:** pointer to the blunker data structure.

**mode:** **mode** determines what estimate values are used for the sequence of corrupt samples to be replaced. 0 = zero mode (estimate as zero); 1 = sample-hold (take the value of non-corrupt signal at the beginning of the impulse and hold that throughout the corrupt sequence); 2 = mean-hold (average the non-corrupt values at the beginning and end of the corrupt sequence and use that as the estimate during the corrupt sequence); 3 = hold-sample (take the value of non-corrupt signal at the end of the impulse and hold that throughout the corrupt sequence); 4 = linearly interpolate across the corrupt sequence.

#### **void pSetRCVRNOBTau (NOB a, double tau)**

**a:** pointer to the blunker data structure.

**tau:** the duration (seconds) of both the **advslewtime** and **hangslewtime** are set to **tau**.

#### **void pSetRCVRNOBHangtime (NOB a, double time)**

**a:** pointer to the blunker data structure.

**time:** hang-time,  $t_{hang}$  (seconds).

#### **void pSetRCVRNOBAdvtme (NOB a, double time)**

**a:** pointer to the blunker data structure.

**time:** advance-time,  $t_{adv}$  (seconds).

#### **void pSetRCVRNOBBacktau (NOB a, double tau)**

**a:** pointer to the blunker data structure.

**tau:** time-constant for averaging power across the wide bandwidth.

#### **void pSetRCVRNOBThreshold (NOB a, double thresh)**

**a**: pointer to the blanker data structure.

**thresh**: detection threshold. (See explanation above in description of 'create' call.)

### **Identifier-Based Calls**

A blanker is instantiated (its resources are allocated and default values set) using the following call.

**NOB create\_nobEXT (int id, int run, int mode, int bufsize, double samplerate, double slewtime, double hangtime, double advtime, double backtau, double threshold)**

**id**: an integer value used as the identifier for this instance of the blanker. The number of blankers is currently restricted (by the setting of an internal variable) to 32. Correspondingly, identifiers must be in the range 0 through 31.

**run**: default value of the **run** variable. 0 = OFF; 1 = ON. When this block is turned OFF, the input buffer is copied to the output buffer in the event they are not the same.

**mode**: default mode. **mode** determines what estimate values are used for the sequence of corrupt samples to be replaced. 0 = zero mode (estimate as zero); 1 = sample-hold (take the value of non-corrupt signal at the beginning of the impulse and hold that throughout the corrupt sequence); 2 = mean-hold (average the non-corrupt values at the beginning and end of the corrupt sequence and use that as the estimate during the corrupt sequence); 3 = hold-sample (take the value of non-corrupt signal at the end of the impulse and hold that throughout the corrupt sequence); 4 = linearly interpolate across the corrupt sequence.

**bufsize**: default number of complex I-Q samples in the input buffer.

**samplerate**: default sample-rate.

**slewtime**: default duration (seconds) of both the slew before advance time and the slew after hang time.

**hangtime**: default hang-time,  $t_{hang}$  (seconds).

**advtime**: default advance-time,  $t_{adv}$  (seconds).

**backtau**: default time-constant for averaging power across the wide bandwidth.

**threshold**: default threshold. A sample is judged to be part of an impulse if its magnitude is greater than **threshold**\***average\_power**.

Resources are freed using the following call.

**void destroy\_nobEXT (int id)**

**id**: identifier for this instance of the blanker.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

Buffers are flushed of stale data and indexes are re-initialized using the following call.

#### **void flush\_nobEXT (int id)**

**id**: identifier for this instance of the blunker.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

A buffer of input data is processed and a buffer of output data written using the following call.

#### **void xnobEXT (int id, double\* in, double\* out)**

**id**: identifier for this instance of the blunker.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer which may be the same as the input buffer.

Various parameters are reset using the following calls. Unless otherwise noted, these following calls may be made at any time including while data is flowing.

#### **void SetEXTNOBRun (int id, int run)**

**id**: identifier for this instance of the blunker.

**run**: 0 = OFF, 1 = ON. When OFF, the input buffer will be copied to the output buffer if they are not the same.

#### **void SetEXTNOBMode (int id, int mode)**

**id**: identifier for this instance of the blunker.

**mode**: **mode** determines what estimate values are used for the sequence of corrupt samples to be replaced. 0 = zero mode (estimate as zero); 1 = sample-hold (take the value of non-corrupt signal at the beginning of the impulse and hold that throughout the corrupt sequence); 2 = mean-hold (average the non-corrupt values at the beginning and end of the corrupt sequence and use that as the estimate during the corrupt sequence); 3 = hold-sample (take the value of non-corrupt signal at the end of the impulse and hold that throughout the corrupt sequence); 4 = linearly interpolate across the corrupt sequence.

#### **void SetEXTNOBSamplerate (int id, int rate)**

**id**: identifier for this instance of the blunker.

**rate**: sample-rate.

**void SetEXTNOBBuffsize (int id, int size)**

**id**: identifier for this instance of the blunker.

**size**: number of complex I-Q samples in the input buffer.

**void SetEXTNOBTau (int id, double tau)**

**id**: identifier for this instance of the blunker.

**tau**: the duration (seconds) of both the `advslewtime` and `hangslewtime` are set to **tau**.

**void SetEXTNOBHangtime (int id, double time)**

**id**: identifier for this instance of the blunker.

**time**: hang-time,  $t_{hang}$  (seconds).

**void SetEXTNOBAdvtme (int id, double time)**

**id**: identifier for this instance of the blunker.

**time**: advance-time,  $t_{adv}$  (seconds).

**void SetEXTNOBBacktau (int id, double tau)**

**id**: identifier for this instance of the blunker.

**tau**: time-constant for averaging power across the wide bandwidth.

**void SetEXTANBNOBThreshold (int id, double thresh)**

**id**: identifier for this instance of the blunker.

**thresh**: detection threshold. (See explanation above in description of 'create' call.)

## Resampler

Resamplers are used to convert from one sample-rate to another sample-rate. Conceptually, the resamplers provided here accomplish this by finding a least common multiple of the input and output frequencies, interpolating up to this least common multiple, and then decimating down from this least common multiple. In the actual computations, care was taken not to perform operations for samples that, in the end, will be discarded anyway.

Resamplers are used within the RXA and TXA channels. However, they are also often useful within various other parts of the application to convert either complex I-Q or simple audio samples from one rate to another.

PROGRAMMING NOTE: There are two sets of calls that can be used to each of the below-described resamplers.

- 1) When calling from the C language, one can include the "resample.h" header file, use a 'create' call that returns a pointer of the native **RESAMPLE** (for the Complex Double-Precision Resampler) or **RESAMPLEF** (Non-Complex Single-Precision Resampler) pointer type, and then use that pointer to identify the resampler data structure in future references.
- 2) When calling from another language, or when it is not desirable to include the "resample.h" header file, there are calls that return and use pointers that have been cast to type **void\***. This was originally done for historical reasons and this author acknowledges that there are potential problems with this approach. With limited exceptions, the C standard does NOT guarantee that pointers to different types will be of the same length or structure. Therefore, pointer type conversions back and forth cannot be guaranteed. This situation is even more questionable when multiple languages are involved. Practically, on common architectures in use and using common tools, this generally works out OK. However, please beware!

### Complex Double-Precision Resampler

As the title implies, this resampler is for complex I-Q samples of double-precision floating-point values.

#### Native Pointers

The resampler is created (its resources allocated), destroyed (its resources freed), flushed (data cleared from buffers and indexes reset), and executed using the following four calls, respectively.

**RESAMPLE create\_resample (int run, int size, double\* in, double\* out, int in\_rate, int out\_rate, double fc, int ncoef, double gain)**

**return value:** The return value is a pointer of type **RESAMPLE**, i.e., a pointer to the resampler data structure that has been created.

**run:** 0 = resampler does not run, it simply copies input buffer (length **size**) to output buffer; 1 = resampler runs. Normally, this parameter should be entered as 1.

**size:** number of complex I-Q samples in the input buffer.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer.

**in\_rate**: input sample-rate (samples / second)

**out\_rate**: output sample-rate (samples / second)

**fc**: cut-off frequency for bandpass filtering. If entered as 0.0, this will be automatically set to  $+/-0.45 * \text{min\_rate}$ , where **min\_rate** = min (**in\_rate**, **out\_rate**). Setting this parameter to 0.0 is generally recommended.

**ncoef**: number of filter coefficients to use. If set to 0, this will be automatically calculated to deliver an excellent filter. Setting this parameter to 0 is generally recommended.

**gain**: a gain factor may be applied in the resampling process, if desired. Otherwise, set this gain to 1.0.

### **void destroy\_resample (RESAMPLE a)**

**a**: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

### **void flush\_resample (RESAMPLE a)**

**a**: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

### **int xresample (RESAMPLE a)**

**return value**: number of samples written to the output buffer. Note that, unless **out\_rate == in\_rate**, the number of output samples will be different than the number of input samples. Also note that each successive buffer of input data MAY NOT produce the same number of output samples.

**a**: pointer to the resampler data structure.

## **void\* Pointers**

These are simplified wrappers around the above calls. The create function returns a void pointer and the void pointer is used to reference the resampler in future calls. Calls to create, destroy, and execute the resampler are as follows, respectively.

### **void\* create\_resampleV (int in\_rate, int out\_rate)**

**return value**: a pointer to the resampler data structure that has been created.

**in\_rate**: input sample-rate (samples / second)

**out\_rate**: output sample-rate (samples / second)

Considering other parameters that are now internal, note that using this simplified approach, fc = 0.0, ncoef = 0, and gain = 1.0 in the creation of the resampling filters.

#### **void destroy\_resampleV (void\* ptr)**

**ptr**: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void xresampleV (double\* in, double\* out, int size, int\* outsamps, void\* ptr)**

**in**: pointer to the input buffer.

**out**: pointer to the output buffer.

**size**: number of complex I-Q samples in the input buffer.

**outsamps**: pointer to location where the number of output samples written to the output buffer is to be stored.

**ptr**: pointer to the resampler data structure.

### **Non-Complex (Audio) Single-Precision Resampler**

As the title implies, this resampler is for non-complex (I-only) samples of single-precision floating-point values.

#### **Native Pointers**

The resampler is created (its resources allocated), destroyed (its resources freed), flushed (data cleared from buffers and indexes reset), and executed using the following four calls, respectively.

#### **RESAMPLEF create\_resampleF (int run, int size, float\* in, float\* out, int in\_rate, int out\_rate)**

**return value**: The return value is a pointer of type **RESAMPLEF**; a pointer to the resampler data structure that has been created.

**run**: 0 = resampler does not run, simply copies input buffer (length **size**) to output buffer; 1 = resampler runs. Normally, this parameter should be entered as 1.

**size**: number of audio (I-only) samples in the input buffer.

**in**: pointer to the input buffer.

**out**: pointer to the output buffer.

`in_rate`: input sample-rate (samples / second)

`out_rate`: output sample-rate (samples / second)

#### **void destroy\_resampleF (RESAMPLEF a)**

`a`: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void flush\_resampleF (RESAMPLEF a)**

`a`: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **int xresampleF (RESAMPLEF a)**

**return value**: number of samples written to the output buffer. Note that, unless `out_rate == in_rate`, the number of output samples will be different than the number of input samples. Also note that each successive buffer of input data MAY NOT produce the same number of output samples.

`a`: pointer to the resampler data structure.

### ***void\* Pointers***

These are simplified wrappers around the above calls. The create function returns a void pointer and the void pointer is used to reference the resampler in future calls. Calls to create, destroy, and execute the resampler are as follows, respectively.

#### **void\* create\_resampleFV (int in\_rate, int out\_rate)**

**return value**: a pointer to the resampler data structure that has been created.

`in_rate`: input sample-rate (samples / second)

`out_rate`: output sample-rate (samples / second)

#### **void destroy\_resampleFV (void\* ptr)**

`ptr`: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void xresampleFV (float\* in, float\* out, int size, int\* outsamps, void\* ptr)**

`in`: pointer to the input buffer.

**out**: pointer to the output buffer.

**size**: number of audio (l-only) samples in the input buffer.

**outsamps**: pointer to location where the number of output samples written to the output buffer is to be stored.

**ptr**: pointer to the resampler data structure.

## Variable Ratio Resampler

When passing sampled signals between clock domains, it is generally necessary to resample with a ratio that is adjustable to a high-level of precision and which cannot necessarily be expressed as the ratio of two integers. An example of this would be audio generated by the radio hardware and subsequent signal processing being passed to a computer sound card to be output to a speaker. The radio has one hardware clock and the sound card has another. Even if the radio output sample-rate is set to, for example, 48Khz, and the sound card input is also set to 48Khz, since the hardware clocks are different, buffer underflows or overflows will eventually result. Resampling with a ratio of, for example 1.0056 or 0.99974 might be necessary to match the rates of the two domains. The Variable Ratio Resampler supports this type resampling where the ratio can be entered as a floating point value and can be changed over time to respond to clock drift.

The Variable Ratio Resampler is instantiated as follows:

```
VARSAMP create_varsamp (int run, int size, double* in, double* out, int in_rate, int out_rate, double fc_high, double fc_low, int R, double gain, double var, int varmode)
```

**return value:** The return value is a pointer of type **VARSAMP**; a pointer to the variable ratio resampler data structure that has been created.

**run:** 0 = resampler does not run, simply copies input buffer (length **size**) to output buffer; 1 = resampler runs. Normally, this parameter should be entered as 1.

**size:** number of double-precision complex (I-Q) samples in the input buffer.

**in:** pointer to the input buffer.

**out:** pointer to the output buffer.

**in\_rate:** nominal input rate, e.g., 48000.

**out\_rate:** nominal output rate, e.g., 192000.

**fc\_high:** cut-off frequency for bandpass filtering. If entered as 0.0, this will be automatically set to  $+/-0.4275 * \text{min\_rate}$ , where **min\_rate** = min (**in\_rate**, **out\_rate**). Setting this parameter to 0.0 is generally recommended.

**fc\_low:** cut-off frequency to create a "notch" around zero Hertz. If set to a negative value, e.g., -1.0, there is NO "notch" around zero Hertz and frequency coverage is continuous from **-fc\_high** to **+ fc\_high**. However, if this is set to a positive value, e.g., 500.0, the spectrum will be "notched" from -500.0 Hertz to +500 Hertz.

**R:** This is the density factor for a table of filter coefficients that is precomputed when this function is called. The "density" determines the ratio of the number of table coefficients to the number of coefficients that are used when the filter executes. For example, if set to 1024, there will be 1024 times as many coefficients in the table as are used for filter execution. Simulation

has shown that 1024 is a good value producing very low spur levels but yet not requiring an excessive amount of memory to store the table.

**gain**: a gain factor may be applied in the resampling process, if desired. Otherwise, set this parameter to 1.0.

**var**: initial value for the variable portion of the resampling ratio. This value should be near 1.0, normally in the range 0.95 to 1.05. Note that the TOTAL resampling ratio is **var** \* (**out\_rate** / **in\_rate**).

**varmode**: If set to 0, the same value of **var** will be used for each entire buffer of data. If set to 1, the value of **var** will be interpolated for each sample beginning with the ending value for the previous buffer of data and ending at the value for the current buffer of data. Setting to 1 therefore produces "smoother" transitions in sample-rate.

Functions to free varsamp resources, flush the current data, and execute processing of a new input buffer of data are as follows:

#### **void destroy\_varsamp (VARSAMP a)**

**a**: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void flush\_varsamp (VARSAMP a)**

**a**: pointer to the resampler data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **int xvarsamp (VARSAMP a, double var)**

**return value**: The number of samples written to the output buffer by the execution of this function.

**a**: pointer to the resampler data structure.

**var**: variable portion of the resampling ratio. A new value can be given for processing each buffer of input data. Note also the comments above regarding **varmode**.

The following four functions would generally be used if the variable rate resampler were integrated into a WDSP unit. Note that data must NOT be flowing when these are executed.

#### **void setBuffers\_varsamp (VARSAMP a, double\* in, double\* out)**

**a**: pointer to the resampler data structure.

**in:** pointer to the input buffer.

**out:** pointer to the output buffer.

**void setSize\_varsamp (VARSAMP a, int size)**

**a:** pointer to the resampler data structure.

**size:** size (number of complex samples) of the input buffer.

**void setInRate\_varsamp (VARSAMP a, int rate)**

**a:** pointer to the resampler data structure.

**rate:** input nominal sample-rate.

**void setOutRate\_varsamp (VARSAMP a, int rate)**

**a:** pointer to the resampler data structure.

**rate:** output nominal sample-rate.

The following two calls are also available to facilitate reconfiguring cutoff frequencies for the variable ratio resampler. Note that data must NOT be flowing when these are executed.

**void setFCLow\_varsamp (VARSAMP a, double fc\_low)**

**a:** pointer to the resampler data structure.

**fc\_low:** cut-off frequency to create a "notch" around zero Hertz. If set to a negative value, e.g., -1.0, there is NO "notch" around zero Hertz and frequency coverage is continuous from **-fc\_high** to + **fc\_high**. However, if this is set to a positive value, e.g., 500.0, the spectrum will be "notched" from -500.0 Hertz to +500 Hertz.

**void setBandwidth\_varsamp (VARSAMP a, double fc\_low, double fc\_high)**

**a:** pointer to the resampler data structure.

**fc\_low:** cut-off frequency to create a "notch" around zero Hertz. If set to a negative value, e.g., -1.0, there is NO "notch" around zero Hertz and frequency coverage is continuous from **-fc\_high** to + **fc\_high**. However, if this is set to a positive value, e.g., 500.0, the spectrum will be "notched" from -500.0 Hertz to +500 Hertz.

**fc\_high:** cut-off frequency for bandpass filtering. If entered as 0.0, this will be automatically set to +/-0.4275 \* **min\_rate**, where **min\_rate** = min (**in\_rate**, **out\_rate**). Setting this parameter to 0.0 is generally recommended.

A simplified version of `create_varsamp( ... )` is provided. Internally, this calls the full version of the function; however, certain parameters have either not been provided at this point or have fixed values. This may be especially useful when the variable ratio resampler is being instantiated and called from another language such as C#.

**void\* create\_varsampV (int in\_rate, int out\_rate, int R)**

**return value:** The return value is a pointer of type `void*`; a pointer to the variable ratio resampler data structure that has been created.

**in\_rate:** input nominal sample-rate.

**out\_rate:** output nominal sample-rate.

**R:** This is the density factor for a table of filter coefficients that is precomputed when this function is called. The "density" determines the ratio of the number of table coefficients to the number of coefficients that are used when the filter executes. For example, if set to 1024, there will be 1024 times as many coefficients in the table as are used for filter execution. Simulation has shown that 1024 is a good value producing very low spur levels but yet not requiring an excessive amount of memory to store the table.

Note that the input buffer size and pointers to the input and output buffers have NOT been specified. They are specified in the subsequent calls to execute the resampler:

**void xvarsampV (double\* input, double\* output, int numsamps, double var, int\* outsamps, void\* ptr)**

**input:** pointer to the input buffer.

**output:** pointer to the output buffer.

**numsamps:** number of double-precision complex (I-Q) samples in the input buffer.

**var:** variable portion of the resampling ratio. A new value can be given for processing each buffer of input data. Note also the comments above regarding **varmode**.

**outsamps:** pointer to a location where the number of samples written to the output buffer is to be stored.

**ptr:** pointer to the resampler data structure.

varsamp resources can also be freed using the `void*` pointer returned from `create_varsampV( ... )`:

**void destroy varsampV (void\* ptr)**

**ptr:** pointer to the resampler data structure.

## Sample-Rate Matcher

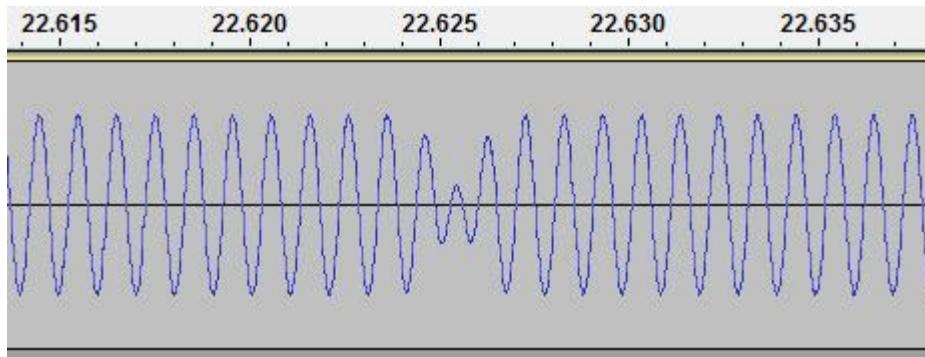
While the Variable Rate Resampler (discussed above) is a key component in "matching" sample-rates between clock domains, it does not solve the entire problem. We must still determine the value of "var", the variable component of the resampling ratio, and we must still buffer data being transferred from one domain to another. The Sample-Rate Matcher comprises an instance of the Variable Rate Resampler and, additionally, these other required functions. Once instantiated, the Sample-Rate Matcher is simply called with data from one domain and called to take out data from the other domain. By buffering the data and measuring the ratio of inputs to outputs it determines "var" and operates the Variable Ratio Resampler.

Due to system distractions/disturbances, occasionally the O/S does not schedule tasks for execution in a timely fashion. In those cases, it may not be practical to eliminate a buffer underflow or overflow. However, the Sample-Rate Matcher contains some additional functionality to mitigate the unpleasantness of those experiences.

In an underflow situation, the internal ring buffer is filled half-full of zeros to avoid a second underflow if there is a successive read before enough data has been written. In this case, the sample-rate matcher "slews" (along a raised-cosine) to zero before the zeros are inserted and then slews back to full volume after the zeros. The waveform looks like this:



In an overflow situation, half of the samples in the internal ring are discarded so that another overflow will not occur if the next write occurs before a sufficient number of samples are read. In that case, the waveform before the discarded samples is "blended" with the waveform after the discarded samples.



Since these actions involve half the samples in the ring buffer, they are substantial and the sound of the disturbance is not totally masked. However, in all cases, discontinuities in the sample stream have been avoided and the high-frequency "pop" energy caused by such discontinuities is eliminated.

The Sample-Rate Matcher is instantiated as follows:

```
RMATCH create_rmatch (int run, double* in, double* out, int insize, int outsize, int nom_inrate, int nom_outrate, double fc_high, double fc_low, double gain, double startup_delay, int auto_ringsize, int ringsize, int R, double var, int ffmav_min, int ffmav_max, double ff_alpha, int prop_ringmin, int prop_ringmax, double prop_gain, int varmode, double tslew)
```

**return value:** The return value is a pointer of type RMATCH; a pointer to the sample-rate matcher data structure that has been created.

**run:** 0 = matcher does not run, it does not read from the input buffer nor does it write to the output buffer. Normally, this parameter should be entered as 1.

**in:** pointer to the input buffer.

**out:** pointer to the output buffer.

**insize:** number of samples to read from the input buffer.

**outsize:** number of samples to write to the output buffer.

**nom\_inrate:** nominal input rate, e.g., 48000.

**nom\_outrate:** nominal output rate, e.g., 192000.

**fc\_high:** cut-off frequency for bandpass filtering. If entered as 0.0, this will be automatically set to  $+/-0.4275 * \text{min\_rate}$ , where **min\_rate** = min (**in\_rate**, **out\_rate**). Setting this parameter to 0.0 is generally recommended.

**fc\_low:** cut-off frequency to create a "notch" around zero Hertz. If set to a negative value, e.g., -1.0, there is NO "notch" around zero Hertz and frequency coverage is continuous from **-fc\_high** to **+ fc\_high**. However, if this is set to a positive value, e.g., 500.0, the spectrum will be "notched" from -500.0 Hertz to +500 Hertz.

**gain:** a gain factor may be applied in the resampling process, if desired. Otherwise, set this parameter to 1.0.

**startup\_delay:** At system startup, it often takes a short time for data to begin flowing. Any control data collected during this time is likely to be erroneous. A non-zero value specified here delays the collection of control data by the amount of time specified (seconds).

**auto-ringsize:** THIS FEATURE HAS NOT BEEN IMPLEMENTED. Set this value to zero.

**ringsize**: the size of the ring buffer (complex samples).

**R**: This is the density factor for a table of filter coefficients that is precomputed when this function is called. The "density" determines the ratio of the number of table coefficients to the number of coefficients that are used when the filter executes. For example, if set to 1024, there will be 1024 times as many coefficients in the table as are used for filter execution. Simulation has shown that 1024 is a good value producing very low spur levels but yet not requiring an excessive amount of memory to store the table.

**var**: initial value for the variable portion of the resampling ratio. This value should be near 1.0, normally in the range 0.95 to 1.05. Note that the TOTAL resampling ratio is **var** \* (**nom\_outrate** / **nom\_inrate**).

**ffmav\_min**: minimum value for the size of the moving average filter used for the feed-forward control component. For usual operating conditions, 4096 is a good value.

**ffmav\_max**: maximum number of data points used for the feed-forward control component. For usual operating conditions, 262144 is a good value. NOTE: This value MUST be a power of two.

**ff\_alpha**: exponential smoothing parameter for the feed-forward control. For usual operating conditions, 0.01 is a good value.

**prop\_ringmin**: minimum value for the size of the moving average filter used for the proportional control component. For usual operating conditions, 4096 is a good value.

**prop\_ringmax**: maximum number of data points used for the proportional control component. For usual operating conditions, 163844 is a good value. NOTE: This value MUST be a power of two.

**prop\_gain**: gain for the proportional feedback term. For usual operating conditions, 4.0e-06 is a good value.

**varmode**: If set to 0, the same value of **var** will be used for each resampler buffer of data. If set to 1, the value of **var** will be interpolated for each sample beginning with the ending value for the previous buffer of data and ending at the value for the current buffer of data. Setting to 1 therefore produces "smoother" transitions in sample-rate.

**slew\_time**: time (seconds) to be used in slewing in the event of underflow or overflow.

Functions to free matcher resources, and to reset the current state are as follows:

**void destroy\_rmatch (RMATCH a)**

**a**: pointer to the matcher data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

#### **void reset\_rmatch (RMATCH a)**

**a**: pointer to the matcher data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

Simplified functions where many parameters are automatically set, and using void\* pointers, are also available to instantiate the Rate Matcher and to free its resources. These may be especially useful when the sample-rate matcher is being instantiated and called from another language such as C#.

#### **void\* create\_rmatchV (int in\_size, int out\_size, int nom\_inrate, int nom\_outrate, int ringsize, double var)**

**return value**: The return value is a pointer of type **void\***; a pointer to the sample-rate matcher data structure that has been created.

**in\_size**: number of samples to read from the input buffer.

**out\_size**: number of samples to write to the output buffer.

**nom\_inrate**: nominal input rate, e.g., 48000.

**nom\_outrate**: nominal output rate, e.g., 192000.

**ringsize**: the size of the ring buffer (complex samples).

**var**: initial value for the variable portion of the resampling ratio.

#### **void destroy\_rmatchV (void\* a)**

**a**: pointer to the matcher data structure.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

The following functions are to, respectively, write data to the Sample-Rate Matcher and read data from the Sample-Rate matcher.

#### **void xrmatchIN (void\* b, double\* in)**

**b**: pointer to the matcher data structure.

**in**: pointer to the input data buffer.

```
void xrmatchOUT (void* b, double* out)
```

**b**: pointer to the matcher data structure.

**out**: pointer to the output data buffer.

Diagnostic information from the Sample-Rate Matcher can be obtained, if desired, using the following function.

```
void getRMatchDiags (void* b, int* underflows, int* overflows, double* var, int* ringsize, int* nring)
```

**b**: pointer to the matcher data structure.

**underflows**: pointer to the location into which the cumulative number of underflows should be stored.

**overflows**: pointer to the location into which the cumulative number of overflows should be stored.

**var**: pointer to the location into which the current value of 'var' should be stored.

**ringsize**: pointer to the location into which the current ring-buffer size should be stored.

**nring**: the current number of samples in the ring buffer.

The 'underflows' and 'overflows' counts can be reset to zero by calling:

```
void resetRMatchDiags (void* b)
```

**b**: pointer to the matcher data structure.

For testing purposes, it may also be desirable to "force" the variable ratio resampler to a particular 'var' ratio. That can be done by calling this function.

```
void forceRMatchVar (void* b, int force, double fvar)
```

**b**: pointer to the matcher data structure.

**force**: 0 = DON'T force; 1 = FORCE.

**fvar**: value of 'var' to force.

As of Revision 1.19, nine additional functions were added to enable dynamic setting and/or checking of various internal parameters for experimentation and tuning purposes. These are:

**void setRMatchFeedbackGain (void\* b, double feedback\_gain)**

**b**: pointer to the matcher data structure.

**feedback\_gain**: this is the ‘prop\_gain’ value as described above in create\_rmatch(...).

**void setRMatchSlewTime (void\* b, double slew\_time)**

**b**: pointer to the matcher data structure.

**slew\_time**: this is the ‘slew\_time’ value as described above in create\_rmatch(...).

**void setRMatchSlewTime1 (void\* b, double slew\_time)**

**b**: pointer to the matcher data structure.

**slew\_time**: this is the ‘slew\_time’ value as described above in create\_rmatch(...).

**void setRMatchFFAlpha (void\* b, double ff\_alpha)**

**b**: pointer to the matcher data structure.

**ff\_alpha**: this is the ‘ff\_alpha’ value as described above in create\_rmatch(...).

**void setRMatchPropRingMin (void\* b, int prop\_min)**

**b**: pointer to the matcher data structure.

**prop\_min**: this is the ‘prop\_ringmin’ value as described above in create\_rmatch(...).

**void setRMatchPropRingMax (void\* b, int prop\_max)**

**b**: pointer to the matcher data structure.

**prop\_max**: this is the ‘prop\_ringmax’ value as described above in create\_rmatch(...).

**void setRMatchFFRingMin (void\* b, int ff\_ringmin)**

**b**: pointer to the matcher data structure.

**ff\_ringmin**: this is the ‘ffmax\_min’ value as described above in create\_rmatch(...).

**void setRMatchFFRingMax (void\* b, int ff\_ringmax)**

**b**: pointer to the matcher data structure.

**ff\_ringmax**: this is the ‘ffmax\_max’ value as described above in create\_rmatch(...).

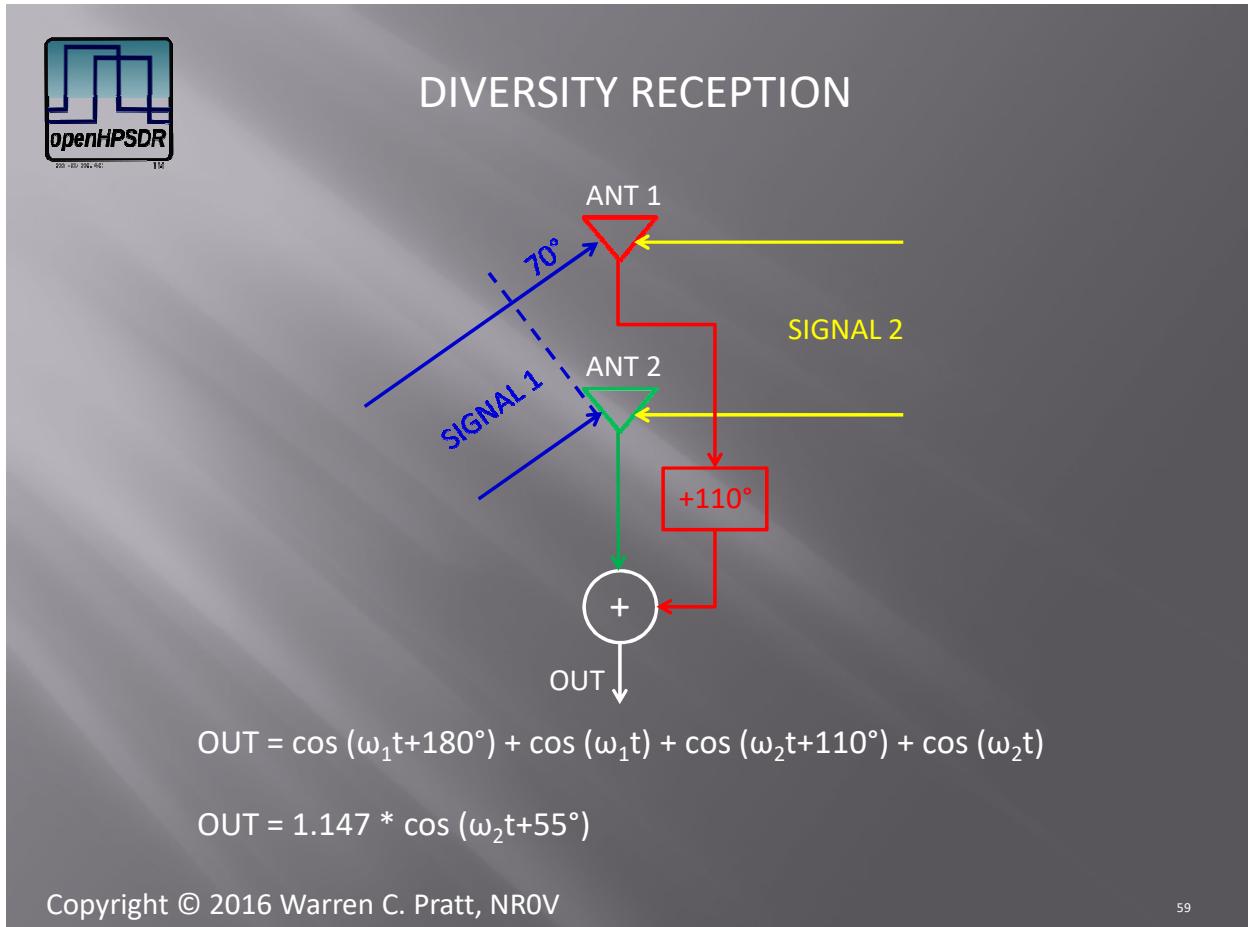
```
void getControlFlag (void* b, int* control_flag)
```

**b**: pointer to the matcher data structure.

**control\_flag**: pointer to location to store ‘control\_flag’.

## Diversity Mixer

The Diversity Mixer is used to implement diversity reception, an important and unique interference reduction technique. Two ADCs, each connected to a separate antenna, are required for this technique. After the signals are digitized by the ADCs, they can be mixed, while applying amplitude and phase adjustments, to null a signal (or noise) that is arriving from a particular direction.



Consider the example shown in the above figure. Signals 1 and 2 are arriving at the antennas from different angles. The path-length (and therefore delay) for Signal 2 is the same to both antennas. However, for Signal 1, the path-length is longer to Antenna 1 which results in an added 70 degrees of phase shift of that signal. Now, suppose we digitize the signals (not shown) and then mix them applying an additional 110 degrees of phase shift to Signal 1. For simplicity, we also assume the signals are of the same magnitude. Looking at the output signal resulting from the mixing (summing), we see that Signal 2 has been slightly strengthened and that Signal 1 is completely absent!

Diversity mixing requires multiple data streams from multiple DDC units. Therefore, it does not fit in the DSP channel model. Diversity mixing is instead performed BEFORE data enters an RXA channel and then the single resulting stream from the mixer is provided as the input to an RXA channel. Note that we could mix more than the two signals shown using more antennas, ADCs, and DDCs.

A diversity mixer is created (its resources are allocated) using the following call.

#### **void create\_divEXT (int id, int run, int nr, int size)**

**id**: an identifier assigned to this diversity mixer. The number of diversity mixers is currently limited to 2 by an internal parameter; so, this must currently be either 0 or 1. This identifier is used to refer to the mixer in subsequent calls.

**run**: default value of run variable. 0 => mixer does not operate, it does copy from **in[0]** to **out**; 1 => inputs are mixed if (**nr == output**), otherwise, **in[output]** is copied to **out**. (See below for setting **output**.)

**nr**: default number of streams to be mixed. The maximum is limited to 8 by an internal parameter.

**size**: default number of I-Q samples in each input buffer.

Resources for the mixer are freed using the following call.

#### **void destroy\_divEXT (int id)**

**id**: identifier for the mixer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

The mixer is flushed using the following call.

#### **void flush\_divEXT (int id)**

**id**: identifier for the mixer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

The mixer executes on the buffers of input data and writes a buffer of output data each time the following call is made.

#### **void xdivEXT (int id, int size, double\*\* in, double\* out)**

**id**: identifier for the mixer.

**size**: number of I-Q samples in each input buffer (and the number that will be written to the output buffer).

**in**: pointer to an array of pointers (**nr** of them) to the input buffers.

**out**: pointer to the output buffer.

The diversity mixer can be turned OFF/ON using:

**void SetEXTDIVRun (int id, int run)**

**id**: identifier for the mixer.

**run**: 0 => mixer does not operate, it does copy from **in[0]** to **out**; 1 => inputs are mixed if (**nr == output**), otherwise, **in[output]** is copied to **out**. (See below for setting **output**.)

The size of the input buffers can be changed using this call:

**void SetEXTDIVBuffsize (int id, int size)**

**id**: identifier for the mixer.

**size**: number of complex I-Q samples in each input buffer.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

The number of streams to be mixed can be changed using:

**void SetEXTDIVNr (int id, int nr)**

**id**: identifier for the mixer.

**nr**: number of streams to be mixed. The maximum is limited to 8 by an internal parameter.

Note: This call must not be executed while data is flowing, i.e., while the execute function is being called.

The **output** value provides a means of having a selected input copied to the **out** buffer rather than running the mixer. if (**output == nr**), the mixer runs. However, if (**output != nr**), **in[output]** is copied to **out**. **output** is set using the following call.

**void SetEXTDIVOutput (int id, int output)**

**id**: identifier for the mixer.

**output**: number of the input stream to copy to the output. if (**output == nr**), the mixer runs.

The I and Q gain/phase-rotation multipliers must be specified for each input stream. Note that I and Q multipliers of 1.0 and 0.0, respectively, may be specified for the "reference receiver."

**void SetEXTDIVRotate (int id, int nr, double\* Irotate, double\* Qrotate)**

**id**: identifier for the mixer.

**nr**: number of input streams

**Irotate**: pointer to vector of I gain/phase-rotation multipliers, one element per input stream.

**Qrotate**: pointer to vector of Q gain/phase-rotation multipliers, one element per input stream.

Before summing, each sample of each stream is modified by its respective I and Q multipliers as follows:

```
I_modified = Irotate[this_stream] * I - Qrotate[this_stream] * Q;
```

```
Q_modified = Irotate[this_stream] * Q + Qrotate[this_stream] * I;
```

## EER

The EER module is present to support power amplifiers operating with modes such as "Envelope Elimination and Restoration" and "Envelope Tracking." Specifically, the EER block accepts the normal I-Q output of the TXA transmitter channel and converts that to two different output streams:

- (1) the original I-Q data multiplied by a scale factor, **mgain**, and
- (2) a choice of one of the following three:
  - a) Phase information only: I-Q phase information multiplied by a scale factor **pgain**. The overall magnitude of the output,  $\sqrt{I^2 + Q^2}$ , is constant at a value of **pgain**; however, I and Q vary in relationship to each other, thereby expressing the phase.
  - b) Magnitude and Phase Information: The original I-Q data multiplied by **pgain**.
  - c) Envelope only: The magnitude of the I-Q envelope, multiplied by **pgain**, and sent only as 'I' values.

When operating in EER/ETR type modes, compensation must be provided for delays in the analog circuitry. Adjustable delays are also included in this software for that purpose.

An EER unit is created (its resources are allocated) using the following call.

**EER create\_eerEXT (int run, int size, double\* in, double\* out, double\* outM, int rate, double mgain, double pgain, int rundelays, double mdelay, double pdelay, int amiq)**

**RETURN VALUE:** A pointer of type **EER** (this pointer is defined in the file "eer.h"). It points to a data structure retaining all the needed information for continuing operation of the EER unit. This pointer will be used in subsequent calls to the EER unit.

**run:** default value of run variable. 0 => EER unit does not operate, it does copy from **in** to **out**; 1 => EER unit operates per the settings specified.

**size:** default number of I-Q samples in each input buffer.

**in:** pointer to input buffer.

**out:** pointer to the output buffer where (a) phase, (b) phase and magnitude, or (c) envelope information are to be stored by the EER unit. Which is stored here will be determined by the setting of the parameter **amiq**. In all cases, the output is scaled by **pgain**. Also, the values may be delayed per EER settings.

**outM:** pointer to an output buffer where the original I-Q input, multiplied by the scale factor **mgain** and possibly delayed, will be stored by the EER unit.

**rate:** the sample-rate of the data.

**mgain:** default multiplier for the **outM** output.

**pgain**: default multiplier for the **out** output.

**rundelays**: 0 => delays will not be used in operation of the EER unit; 1 => delays will be activated in operation of the EER unit.

**mdelay**: default delay (us) for **outM** output.

**pdelay**: default delay (us) for **out** output.

**amiq**: default for what is to be provided on the **out** output.

0: Phase information only: I-Q phase information multiplied by a scale factor **pgain**.

The overall magnitude of the output,  $\sqrt{I^2 + Q^2}$ , is constant at a value of **pgain**; however, I and Q vary in relationship to each other, thereby expressing the phase.

1: Magnitude and Phase Information: The original I-Q data multiplied by **pgain**.

2: Envelope only: The magnitude of the I-Q envelope, multiplied by **pgain**, and sent only as 'I' values.

The EER unit can be turned OFF/ON using the call:

**void pSetEERRun (EER a, int run)**

**a**: pointer of type **EER** to the EER unit.

**run**: 0 => EER does not operate, it does copy from **in[0]** to **out**; 1 => EER is operational.

The size of the input buffers can be changed using this call:

**void pSetEERSize (EER a, int size)**

**a**: pointer of type **EER** to the EER unit.

**size**: number of complex I-Q samples in each input buffer.

The samplerate in use for input and output data can be changed using this call:

**void pSetEERSamplerate (EER a, int rate)**

**a**: pointer of type **EER** to the EER unit.

**rate**: the samplerate of the input and output data.

The gain to be applied to the **outM** output can be changed using this call:

**void pSetEERMgain (EER a, double gain)**

**a**: pointer of type **EER** to the EER unit.

**gain**: the gain (linear scale) to be applied to the **outM** output.

The gain to be applied to the **out** output can be changed using this call:

**void pSetEERPgain (EER a, double gain)**

**a**: pointer of type **EER** to the EER unit.

**gain**: the gain (linear scale) to be applied to the **out** output.

The use of delays can be turned OFF/ON using the call:

**void pSetERRRunDelays (EER a, int run)**

**a**: pointer of type **EER** to the EER unit.

**run**: 0 => delays do not operate; 1 => delays are operational.

The following call is used to set the delay to be used for the **outM** output data:

**void pSetEERMdelay (EER a, double delay)**

**a**: pointer of type **EER** to the EER unit.

**delay**: delay (us). Note that fractional us may be used.

The following call is used to set the delay to be used for the **out** output data:

**void pSetEERPdelay (EER a, double delay)**

**a**: pointer of type **EER** to the EER unit.

**delay**: delay (us). Note that fractional us may be used.

The following call is used to set the mode for the **out** output. See descriptions above for the various options.

**void pSetEERAMIQ (EER a, int amiq)**

**a**: pointer of type **EER** to the EER unit.

**amiq**: output type to be used, per above description of options.

## Combined VOX & Downward Expander

This block is an integrated implementation of VOX and a Downward Expander. They are integrated in that the same triggering is used for both and their operation is synchronized in time. Several advanced features are incorporated such as a 'side-channel filter' to select a range of speech frequencies for triggering and a 'look ahead algorithm' to avoid clipping the first sounds of a speech segment. Anti-VOX functionality, allowing received audio to reduce VOX sensitivity, is also provided.

The software author wishes to thank Scott, WU2O, for his input in defining the functionality of this block.

This block is instantiated using the following call:

```
void create_dexp (int id, int run_dexp, int size, double* in, double* out, int rate, double dettau,
double tattack, double tdecay, double thold, double exp_ratio, double hyst_ratio, double
attack_thresh, int nc, int wtype, double lowcut, double highcut, int run_filt, int run_vox, int
run_audelay, double audelay, void (__stdcall *pushvox)( int id, int active) , int antivox_run, int
antivox_size, int antivox_rate, double antivox_gain, double antivox_tau)
```

**id**: The identifier for this instance of the block. A value from 0 to 3.

**run\_dexp**: 0 to turn OFF the downward expander; 1 to turn ON the downward expander.

**size**: Buffer size (complex samples) of the input and output buffers. This value must be a power-of-two, must be less than or equal to 'nc', and must be less than 'rate'.

**in**: Pointer to the input buffer.

**out**: Pointer to the output buffer which may be the same as the input buffer if desired.

**rate**: Sample-rate.

**dettau**: Time-constant used for low-pass filtering of the input samples to the detector (seconds).

**tattack**: Attack time. This is the raised-cosine length (in seconds) from low-gain to high-gain of the downward expander.

**tdecay**: Release/decay time. This is the raised-cosine length (in seconds) from high-gain to low-gain of the downward expander.

**thold**: Hold time. Time (in seconds) from when the trigger signal drops below the threshold to initiation of the release raised-cosine gain curve.

**exp\_ratio**: Expansion ratio. The ratio between full-gain (gain = 1.0) and the reduced gain level for the downward expander. Note that this is on a linear scale, i.e., if you want to express this in dB in your User Interface, you must then convert from dB to linear scale.

**hyst\_ratio**: Hysteresis ratio. This is a ratio (< 1.0) between (1) the level at which VOX/DEXP trigger and (2) the level at which hold-time commences for VOX/DEXP.

**attack\_thresh**: The trigger value at which VOX is triggered and DEXP attack commences.

**nc**: Number of coefficients of the side-channel filter. This must be a power-of-two and an integral multiple of the buffer size, 'size'.

**wtype**: Window type to be used in generation of the side-channel filter. 0 for 4-term Blackman-Harris and 1 for 7-term Blackman-Harris.

**lowcut**: Low-end cutoff frequency (Hertz) of the side-channel trigger filter.

**highcut**: High-end cutoff frequency (Hertz) of the side-channel trigger filter.

**run\_filt**: 0 to turn OFF the side-channel trigger filter; 1 to turn ON the side-channel trigger filter.

**run\_vox**: 0 to turn OFF VOX operation; 1 to turn ON VOX operation.

**run\_audelay**: 0 to turn OFF the 'lookahead' delay; 1 to turn ON the 'lookahead' delay.

**audelay**: Lookahead delay (seconds).

**(\_\_stdcall \*pushvox)( int id, int active)**: Pointer to the function to call to change the state of the transmitter based upon VOX information. 'Active' is 1 to turn ON the transmitter; 'Active' is 0 to turn OFF the transmitter.

**antivox\_run**: 0 to turn OFF anti-vox and 1 to turn ON anti-vox.

**antivox\_size**: Size (complex samples) of each buffer of anti-vox data.

**antivox\_rate**: Sample-rate of anti-vox data.

**antivox\_gain**: Gain value to be applied to anti-vox data.

**antivox\_tau**: Time-constant to be applied in smoothing anti-vox data.

To release all resources of the block, use the call:

**void destroy\_dexp (int id)**

**id**: The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

To flush stale data from an instance of this block, call:

**void flush\_dexp (int id)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

To execute the block when a new buffer of input data is available, call:

#### **void xdexp (int id)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

If the pointer to the VOX function was not known when 'create\_dexp' was called, or if the function to call needs to be changed after creation, the following function can be called to do so:

#### **void SendCBPushDexpVox (int id, (\_\_stdcall \*pushvox)( int id, int active))**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**(\_\_stdcall \*pushvox)( int id, int active):** Pointer to the function to call to change the state of the transmitter based upon VOX information. 'Active' is 1 to turn ON the transmitter; 'Active' is 0 to turn OFF the transmitter.

The downward expander can be turned OFF/ON using the following call. Note that VOX can be used with or without the downward expander running.

#### **void SetDEXPRun (int id, int run)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**run:** 0 to turn OFF the downward expander; 1 to turn ON the downward expander.

Use the following call to set/reset the input/output buffer size. This value must be a power-of-two, must be less than or equal to 'nc', and must be less than the value of 'rate'.

#### **void SetDEXPSize (int id, int size)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**size:** Buffer size (complex samples) of the input and output buffers.

The following call can be used to set/reset the input and output buffer pointers. Note that the output buffer can be the same as the input buffer if desired.

#### **void SetDEXPIOBuffers (int id, double\* in, double\* out)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**in:** Pointer to the input buffer.

**out:** Pointer to the output buffer.

The sample rate in use can be changed using the following call:

**void SetDEXPRate (int id, int rate)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**rate:** Sample-rate.

The following call is used to set/reset the time-constant of the trigger detector:

**void SetDEXPDetectorTau (int id, double tau)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**tau:** Time-constant used for low-pass filtering of the input samples to the detector (seconds).

The DEXP Attack Time can be adjusted using the following call:

**void SetDEXPAccackTime (int id, double time)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**time:** Attack time. This is the raised-cosine length (in seconds) from low-gain to high-gain of the downward expander.

The DEXP Release Time is adjusted using the following call:

**void SetDEXPReleaseTime (int id, double time)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**time:** Release/decay time. This is the raised-cosine length (in seconds) from high-gain to low-gain of the downward expander.

The DEXP/VOX Hold Time can be adjusted as follows:

**void SetDEXPHoldTime (int id, double time)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**time:** Hold time. Time (in seconds) from when the trigger signal drops below the threshold to initiation of the release raised-cosine gain curve.

The DEXP Expansion Ratio is adjusted as follows:

**void SetDEXPExpansionRatio (int id, double ratio)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**ratio:** Expansion ratio. The ratio between full-gain (gain = 1.0) and the reduced gain level for the downward expander. Note that this is on a linear scale, i.e., if you want to express this in dB in your User Interface, you must then convert from dB to linear scale.

To adjust the DEXP Hysteresis Ratio:

**void SetDEXPHysteresisRatio (int id, double ratio)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**ratio:** Hysteresis ratio. This is a ratio (< 1.0) between (1) the level at which VOX/DEXP trigger and (2) the level at which hold-time commences for VOX/DEXP.

The Attack Threshold is adjusted as follows:

**void SetDEXPAccackThreshold (int id, double thresh)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**thresh:** The trigger value at which VOX is triggered and DEXP attack commences.

The number of filter taps used for the side-channel filter can be changed after creation using the following call:

**void SetDEXPFilterTaps (int id, int taps)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**taps:** Number of coefficients of the side-channel filter. This must be a power-of-two and an integral multiple of the buffer size, 'size'.

To set the window-type to be used in generation of the side-channel filter:

### **void SetDEXPWindowType (int id, int type)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**type:** Window type to be used in generation of the side-channel filter. 0 for 4-term Blackman-Harris and 1 for 7-term Blackman-Harris.

Use the following calls to set the 'low cut' and 'high cut' for the side-channel trigger filter:

### **void SetDEXPLowCut (int id, double lowcut)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**lowcut:** Low-end cutoff frequency (Hertz) of the side-channel trigger filter.

### **void SetDEXPHighCut (int id, double highcut)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**highcut:** High-end cutoff frequency (Hertz) of the side-channel trigger filter.

The side-channel trigger bandpass filter can be turned OFF/ON using the following call. Note that VOX and DEXP can be used with or without this filter.

### **void SetDEXPRunSideChannelFilter (int id, int run)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**run:** '0' turns the side-channel filter OFF; '1' turns the side channel filter ON.

VOX can be turned OFF/ON using the following call. Note that the DEXP can be used with or without VOX being active.

### **void SetDEXPRunVox (int id, int run)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**run:** '0' turns VOX OFF; '1' turns VOX ON.

The following call turns the audio look-ahead OFF/ON. This can be turned OFF/ON independent of whether VOX or DEXP are active.

### **void SetDEXPRunAudioDelay (int id, int run)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**run:** 0 to turn OFF the 'lookahead' delay; 1 to turn ON the 'lookahead' delay.

To set the amount of audio delay (look-ahead), use the following call:

**void SetDEXPAudioDelay (int id, double delay)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**delay:** Lookahead delay (seconds).

For console monitoring purposes, the peak signal value can be obtained using the following call:

**void SetDEXPPeakSignal (int id, double\* peak)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**peak:** Pointer to where the peak signal value is to be stored.

Anti-vox functionality is turned OFF/ON with the following call:

**void SetAntiVOXRun (int id, int run)**

**id:** The identifier initially specified to create this instance of the VOX/DEXP block. (An integer of 0 through 3).

**run:** 0 to turn OFF anti-vox functionality; 1 to turn ON anti-vox functionality.

The size (in complex samples) of each buffer of anti-vox data to be passed to the anti-vox algorithm is set as follows:

**void SetAntiVOXSize (int id, int size)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**size:** The number of complex samples per buffer of anti-vox data.

The sample-rate of the anti-vox data is specified using the following call:

**void SetAntiVOXRate (int id, int rate)**

**id:** The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**rate**: The sample-rate of the anti-vox data.

The gain to be applied to the anti-vox data is set as follows:

**void SetAntiVOXGain (int id, double gain)**

**id**: The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**gain**: Gain (linear scale) to be applied to the anti-vox data.

The time-constant (seconds) used for smoothing the anti-vox data is specified using this call:

**void SetAntiVOXDetectorTau (int id, double tau)**

**id**: The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**tau**: The time constant (seconds) used for smoothing anti-vox data.

Each buffer of anti-vox data is sent to the anti-vox algorithm using the following call. Note that the parameter 'nsamples' is specified, for compatibility reasons. HOWEVER, this parameter is NOT used. The number of samples being transferred is specified in create\_dexp(...) and/or in SetAntiVOXSize(...).

**void SendAntiVOXData (int id, int nsamples, double\* data)**

**id**: The identifier initially specified to create this instance of the block. (An integer of 0 through 3).

**size**: The number of complex samples per buffer of anti-vox data.

## Bandpass Filter Characterization Utility

This block provides the ability to get a magnitude characterization of a bandpass filter. This characterization can be used for graphical, to illustrate the filter, or otherwise.

Resources for a BFCU (bandpass filter characterization utility) must be allocated before its use. Note that up to four instances may be allocated and in use at a time. They are given IDs of 0 – 3. The available number could be extended in the code if needed.

```
int create_bfcu (int id, int min_size, int max_size, double rate, double corner, int points)
```

**id**: The identifier specified to create this instance of the block. (An integer of 0 – 3.)

**min\_size**: The minimum impulse response size that will be used for the filter, i.e., minimum filter size. This MUST be a power of 2.

**max\_size**: The maximum impulse response size that will be used for the filter, i.e., maximum filter size. This MUST be a power of 2.

**rate**: The sample-rate at which the filter operates.

**corner**: The -6dB corner frequency. NOTE: As created, the bandpass filter will be symmetrical about zero, i.e., the corners will be at '+corner' and '-corner'.

**points**: The number of points to generate for each filter response. This value (1) must be a power of 2, and (2) must be  $\geq \text{max\_size}$ .

After use, the resources consumed by the BFCU should be released as follows:

```
void destroy_bfcu (int id)
```

**id**: The identifier specified to create this instance of the block. (An integer of 0 – 3.)

For convenience in using the results of this utility, the following returns the indexes (indexes within the array to be returned) of the upper and lower corners of the filter characterization. The values are stored at 'lower-index' and 'upper-index'.

```
void getFilterCorners (int id, int* lower_index, int* upper_index)
```

**id**: The identifier specified to create this instance of the block. (An integer of 0 – 3.)

**lower\_index**: Pointer to location for lower index value.

**upper\_index**: Pointer to location for upper index value.

Finally, a call to retrieve the actual curve/results of the filter characterization.

```
void getFilterCurve (int id, int size, int w_type, int index_low, int index_high, double* segment)
```

**id**: The identifier specified to create this instance of the block. (An integer of 0 – 3.)

**size**: The current impulse response size (filter size).

**w\_type**: The window-type used for the filter. Currently supported are '0' for Blackman-Harris 4 and '1' for Blackman-Harris 7.

**index\_low**: The lower index of the segment you want returned. The range is '0' through 'points-1'.

**index\_high**: The upper index of the segment you want returned. The range is '0' through 'points-1'.

**segment**: Pointer to location where the results are to be stored.

## Filter Impulse Response Cache

Storing impulse responses for filters that have been previously used is a mechanism that can be used to speed-up things such as mode changes or changes in filter parameters. When available, a filter response stored in the cache will be used rather than re-creating it. Note that the memory used to store impulse responses, and optionally the file-size used when the cache is saved between application uses, can become quite large.

Call this function to initialize the cache.

### `void init_impulse_cash (int use)`

**use:** Set to ‘1’ to use the cache, ‘0’ otherwise.

After the impulse cache is initialized, call this to turn it ON or OFF.

### `void use_impulse_cash (int use)`

**use:** Set to ‘1’ to use the cache, ‘0’ otherwise.

Call to free resources used by the impulse cache.

### `void destroy_impulse_cash (void)`

**No parameters.**

Call to write impulse cache to a file.

### `void save_impulse_cash (const char* path)`

**path:** File path, including filename.

Call to read impulse cache from a file.

### `void read_impulse_cash (const char* path)`

**path:** File path, including filename.

## WDSP Version Information

The calling application may wish to query WDSP for its version number. This allows enabling or disabling certain console features based upon which version of WDSP is in use. It also enables checking to be sure that an appropriate version of WDSP is found.

The following function is available for this purpose.

**int GetWDSPVersion ()**

As written in text, WDSP version numbers always contain exactly two digits to the right of the decimal point. However, for easy comparison with expected version values, the value returned by this function is `version_number*100`. E.g., for Version 1.14, the function will return 114.

## Revision History

<b>Rev</b>	<b>Date</b>	<b>Changes</b>
0.10	2016-08-05	Initial version of this document; released for comments.
0.11	2016-08-24	Added TXASetSipSpecmode(...) in WDSP and in this document. Added comments on SetPSControl().
0.12	2016-11-28	Added new functions calls for PureSignal 2.0.
1.14	2017-03-17	Added GetWDSPVersion(). Added 1 to the revision number of this document such that it now matches the WDSP revision. Documented calls for EER.
1.15	2017-05-06	Added AM_LSB and AM_USB transmit modes. Added CFC and Phase Rotator audio tools.
1.16	2017-05-28	Added CFC Metering. Moved phase rotator in processing chain.
1.17	2017-08-25	Added Variable Ratio Resampler and Rate Matching in WDSP and document.
1.18	2019-01-21	Added 'dexp', combined VOX and Downward Expander.
1.19	2022-01-27	Added function to get CFC data to display compression vs frequency. For the panadapter/waterfall, added an 'rms' detector type. For the variable ratio resampler, added 9 functions to allow dynamically setting/getting some internal parameters for experimentation and tuning purposes. Added additional arguments in functions getRMatchDiags(...) and create_rmatchV(...).
1.20	2022-02-15	Internal enhancements supporting portability to non-Windows operating systems. Enhanced panadapter/waterfall frequency accuracy for high-zoom cases. Porting Considerations: (1) There are two new files that must be included in the build - calculus.c and calculus.h, (2) for the display, setup for calls to SetAnalyzer(...) is somewhat different for optimum results and parameters fscLin and fscHin should be passed as doubles -- see this Guide.
1.21	2023-04-29	Added a volume limiter to the FM Demodulator to balance signal and noise volume during squelch tail or weak signal reception with squelch OFF. Added Voice Squelch functionality as an alternative especially for SSB reception.
1.22	2023-05-20	Efficiency Improvements - less system calls at the expense of slightly higher average thread and memory allocations.
1.23	2023-06-17	Efficiency Improvements.
1.24	2023-11-08	Added time adjustment for TXA carrier up-slew. Added bandwidth adjustment for FM transmit and receive. Added a function that can be used to clear display buffers. In the phase rotator, added a function to reverse microphone phase (180 degrees).

<b>Rev</b>	<b>Date</b>	<b>Changes</b>
1.25	2024-07-05	Added "Trained" mode for Spectral Noise Reduction along with NSTAT NPE Method.
1.26	2025-06-30	Added Bandpass Characterization Utility to provide a graphical characterization of bandpass filter magnitude. Added Filter Impulse Response Caching to speed-up re-use of previously used filters. Removed requirement to use an external file, zetaHat.bin, for Trained NR2 mode. For TX, added pulsed-tone and pulsed-two-tone capability. Added capability to return the largest bin value from a specified band within the display FFT output.
1.27	2025-09-19	Various code changes to make the CLANG LLVM Static Analyzer checker tool (used for Linux implementations) happier. Addition of psychoacoustic post-processing for NR2 Noise Reduction. Minimum Phase filter responses were improperly cached - this has been removed.
1.28	2025-10-21	Minimum Phase filter caching resolved and added back. Added impulse response generation for Gaussian filters for future testing and potential inclusion in the receiver.
1.29	2026-01-16	To the existing CW APF, added new filter options: DoublePole, Matched, and Gaussian.