openHPSDR

ChannelMaster Guide

For Software Developers

The ChannelMaster provides an entire data-path and data-architecture for the openHPSDR radios, specifically using the new ethernet protocol. This guide explains its structure and the basics of its application.

Dr. Warren C. Pratt, NR0V
Revision 0.1
openHPSDR

Table of Contents

Introduction

The ChannelMaster has been written by Doug Wigley, W5WC, and Warren Pratt, NR0V, as an architecture and a body of code serving as the data-path and command-and-control network layer for radios utilizing the new openHPSDR ethernet protocol.

Per the definition of the new ethernet protocol, all inbound data is received from the radio in ethernet packets addressed to specific ports and all outbound data is sent to the radio in ethernet packets addressed to specific ports. The port numbers reveal the type of data contained in the various packets. The ChannelMaster provides a framework to decode and process inbound packets and to form and send the outbound packets. In doing so, the ChannelMaster utilizes other software components, for example, WDSP processing channels. The ChannelMaster is a complete, functional, body of code for this purpose as implemented in Thetis. It can also be viewed as an "architecture" in that some of the functional units, for example the VAC interface or WAV player, might be different in other radio consoles. However, the same structure can still be used just by modifying the code to instantiate and execute the alternative functional units.

The ChannelMaster instantiates and executes various functional blocks -- for example, the DSP channels and VAC interfaces. However, the console software still communicates directly with these blocks for control purposes, with no intervention of the ChannelMaster. For example, the console might request 7 "receivers". The ChannelMaster will instantiate those and make sure the appropriate data is routed to them for execution. However, the console communicates directly with these "receivers" to set modes of operation, filters, WAV recording options, VAC parameters, etc.

The ChannelMaster unpacks Command-and-Control packets from the radio hardware and makes the information available to the console. It also accepts settings from the console, packs it appropriately into Command-and-Control packets, and gets those on their way to the radio hardware.

This architecture is relatively new and the authors can already see ways to improve it and make its operations and interactions with the console more uniform. While no major structural changes are anticipated, the code is likely to evolve over time, as development time permits.

Data Flow

Following The Data Through The Code

Packets of data from the radio arrive to the ChannelMaster in the function ReadThreadMainLoop() in network.c. ReadUDPFrame(), also in network.c, provides additional processing for some types of packets. ReadThreadMainLoop() converts integer DDC I-Q samples and MIC samples to doubles and sends the data on its way using appropriate calls.

DDC Samples

For DDC samples, xrouter() in the file router.c is called with a pointer to the samples and their source port number for identification. The router's purpose is to get the data from each DDC to the right place under the right conditions. For example, data from a particular DDC may go to be processed by a Receiver (see section on "Receiver Data Flow") during receive but may be directed to the Transmitter for PureSignal calibration during transmit. Likewise, for example, if diversity reception is enabled, data from multiple DDCs would be routed to the diversity mixer; however, if it is not enabled, it would go other places. Based upon a router-table and the state of several variables, the router makes real-time decisions on where to send the data and gets it on its way. A packet of DDC data is one of two types: (1) data from a single DDC, or (2) interleaved data from multiple DDCs.

If the packet is of the first type, the router makes a call to Inbound() in cmbuffs.c, with a pointer to the samples and an identifier indicating to which software receiver the samples are to go. Inbound() puts the data in a buffer and, when sufficient data is available for processing by the software receiver, cm_main() [cmbuffs.c] pulls the data from the buffer using cmdata() [cmbuffs.c] and then calls xcmaster() [cmaster.c] with the identifier of the particular receiver.

If the packet is of the second type, the router de-interleaves the data and separates the multiple DDC streams into separate arrays. It then calls InboundBlock() [sync.c] with an appropriate identifier and an array of pointers to the DDC data arrays. Depending upon the identifier, InboundBlock() will either:

- make a call to the diversity mixer xdivEXT() in WDSP and follow that with a called to Inbound() [cmbuffs.c] to pass along the mixed data to be buffered, as described above, and then passed to xcmaster() [cmaster.c],
- make a call to pscc() in WDSP to pass the data to PureSignal for calibration purposes,
- make two calls to Inbound() [cmbuffs.c] to pass data from two DDCs bound for two software receivers, or
- make a single call to Inbound() [cmbuffs.c] to pass one of the sets of data bound for a software receiver while discarding any other data sets.

For more information on the router, see the section on "The Router".

For discussion of receiver data processing in xcmaster(), see the section on "Receiver Data Flow".

MIC Samples

For MIC samples, Inbound() in cmbuffs.c is called with a pointer to the samples and an identifier indicating this is MIC data bound for the transmitter. Inbound() puts the data in a buffer and, when sufficient data is available for processing by the software transmitter, cm_main() [cmbuffs.c] pulls the data from the buffer using cmdata() [cmbuffs.c] and then calls xcmaster() [cmaster.c] with the identifier of the transmitter.

For discussion of transmitter data processing in xcmaster(), see the section on "Transmitter Data Flow".

Audio Output (To Radio)

As shown in the section "Summary of Outbound Data (To Radio)," audio samples from the output of the global asynchronous audio mixer need to make their way back to the radio. When the mixer has prepared a sufficient number of audio samples, mix_main() [aamix.c] calls OutBound() [obbuffs.c] with a pointer to the samples and an identifier to indicate they are audio. OutBound() stores the samples in a buffer. When there are enough samples to fill a packet, ob_main() [obbuffs.c] retrieves samples using obdata() [obbuffs.c] and calls sendOutbound() [network.c]. SendOutbound() rounds and converts the samples to integers and calls WriteUDPFrame() [network.c] with a pointer to the samples and the 'audio' identifier. SendPacket() [network.c] then gets them on their way to the 'audio' port.

TX I-Q Output (To Radio)

As shown in the section "Summary of Outbound Data (To Radio)," TX I-Q samples come from the interleaver. xilv() [ilv.c] calls OutBound() [obbuffs.c] with a pointer to the samples and an identifier to indicate that they are TX I-Q. OutBound() stores the samples in a buffer. When there are enough samples to fill a packet, ob_main() [obbuffs.c] retrieves samples using obdata() [obbuffs.c] and calls sendOutbound() [network.c]. SendOutbound() rounds and converts the samples to integers and calls WriteUDPFrame() [network.c] with a pointer to the samples and the 'TX I-Q' identifier. SendPacket() [network.c] then gets them on their way to the 'TX I-Q' port.

Wideband Display Data Flow

Wideband data is processed and sent to the appropriate display unit(s) in ReadUDPFrame() in the file "network.c". Samples are converted from integers to doubles and there is a simple state machine that manages sending the data to the appropriate display unit. The state machine looks for the beginning of a complete frame of data, starts from there, sending subsequent packets to complete the frame. If a sequence error is encountered, the frame is padded to the end with zeros and the state machine waits for the next frame to start.

Receiver Data Flow

The orchestration of the data processing for receivers and transmitters is found in the files "cmaster.c" and "pipe.c". It is here that the various functional blocks shown in the RX1 RECEIVER, RX2 RECEIVER and TRANSMITTER block diagrams that follow are instantiated, called for execution, and destroyed when they are no longer needed. The console resources (specifically types of supported displays) are slightly different in Thetis for RX1 and RX2; this explains why two separate diagrams are provided.

With regard to the two files "cmaster.c" and "pipe.c", the standard functional blocks that will always be used and are expected to be the same in all radio consoles have been managed in "cmaster.c". In contrast, functional blocks that are more likely to be replaced with alternates in various radio consoles have been broken out and managed in "pipe.c".

Note there is no constraint to have only one "RX1 Receiver" or only one "RX2 Receiver." The ChannelMaster can instantiate and execute as many of either type as desired. The initial Thetis console, however, only has controls for one of each. Also, to be clear, when one of these RXn Receivers is requested/instantiated, EACH of the blocks shown in the diagrams below is created for it.

Comments on the various functional blocks follow.

WAV Player: One per RXn Receiver. Instantiated/owned by the receiver. WAV Player code was written originally in C# and has not been translated to C. Therefore, calls to instantiate and execute WAV Players are made to methods in C#.

WAV Recorder: One per RXn Receiver. Instantiated/owned by the receiver. WAV Recorder code was written originally in C# and has not been translated to C. Therefore, calls to instantiate and execute WAV Recorders are made to methods in C#.

Phase2 Display: One per RX1 Receiver (not used in the RX2 Receiver-type). Instantiated/owned by the receiver. Phase2 Display code was written originally in C# and has not been translated to C. Therefore, calls to instantiate and execute Phase2 Displays are made to methods in C#.

VAC Interface: One interface for inbound data and one interface for outbound data per RXn Receiver. Instantiated/owned by the receiver. The VAC interface is written in C and is found in the files "ivac.c" and "ivac.h".

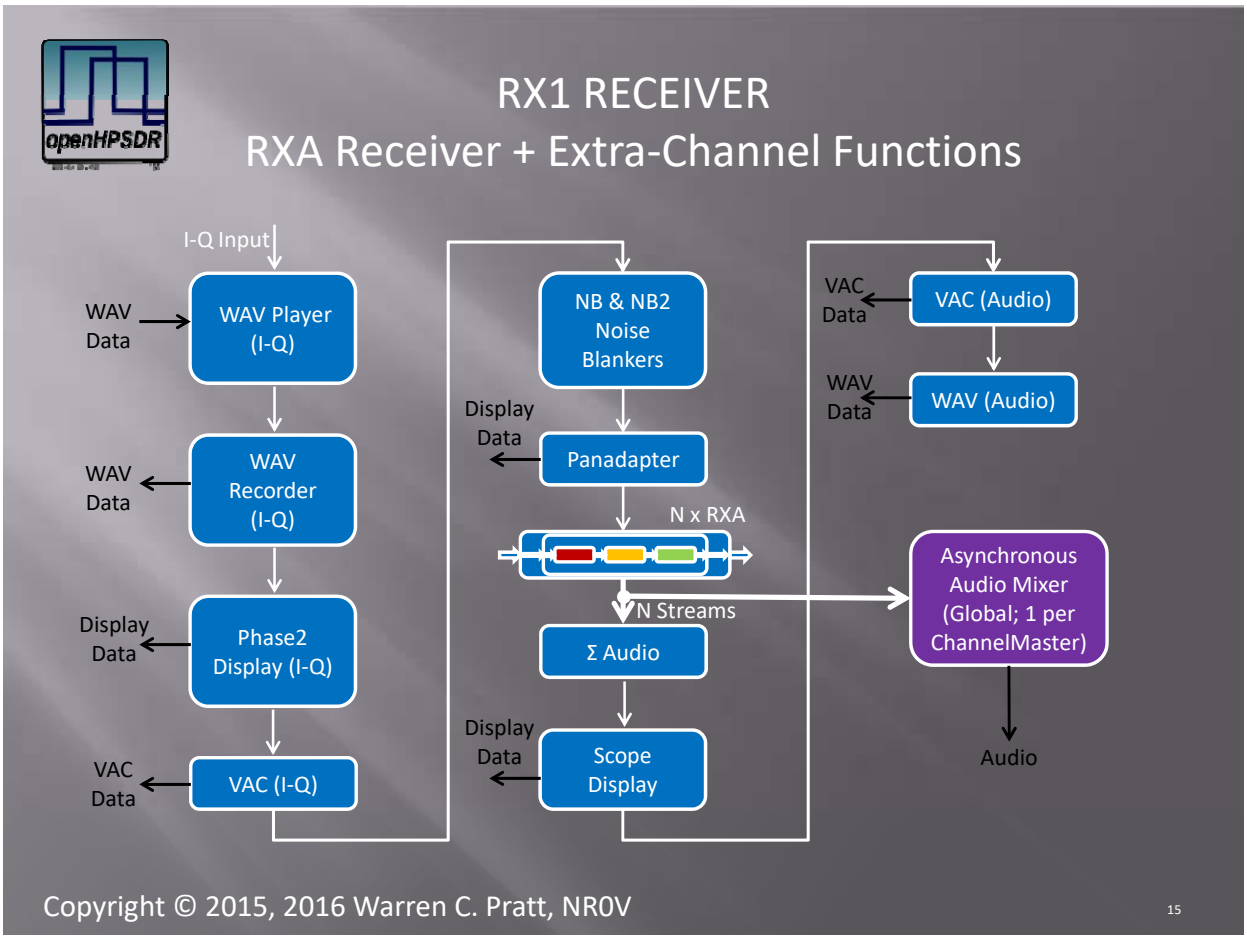
NB & NB2 Noise Blankers: One of each for each RXn Receiver. Instantiated/owned by the receiver. The code for these blankers is part of the wdsp.dll. However, control interfaces to that code are found in files "znob.c", "znob.h", "znobII.c", and "znobII.h".

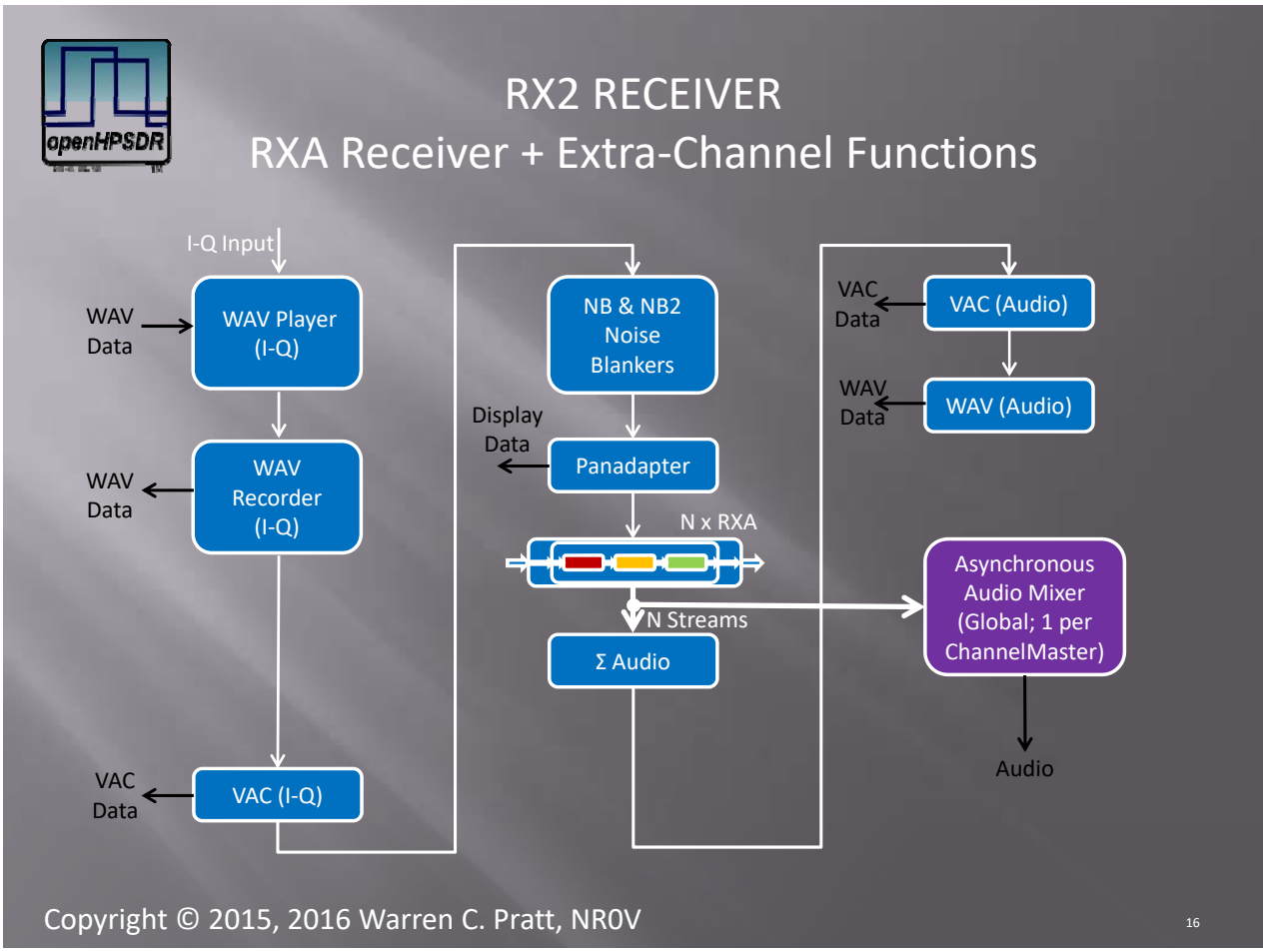
Panadapter: One for each RXn Receiver. Instantiated/owned by the receiver. Per-pixel dB amplitude values are separately computed for a panadapter display and for a waterfall display. Graphics code to draw the displays is part of the console.

RXA DSP Channel: See the "WDSP Guide" for information on all the functions that comprise an RXA Channel Unit. Note that there are N of these instantiated/owned by the RXn Receiver, i.e., there is one

per sub-receiver. In WDSP, there is no distinction between a "main-receiver" and a "sub-receiver". Therefore, in the case of Thetis, for example, N = 2 for the RX1 Receiver. Audio outputs of these RXA channels are combined for further proecssing.

Scope Display: One per RX1 Receiver (not used in the RX2 Receiver-type). Instantiated/owned by the receiver. Scope Display code was written originally in C# and has not been translated to C. Therefore, calls to instantiate and execute the Scope Display are made to methods in C#.





Transmitter Data Flow

It is currently assumed that there is only one transmitter. However, this could be changed relatively easily if needs for more arise.

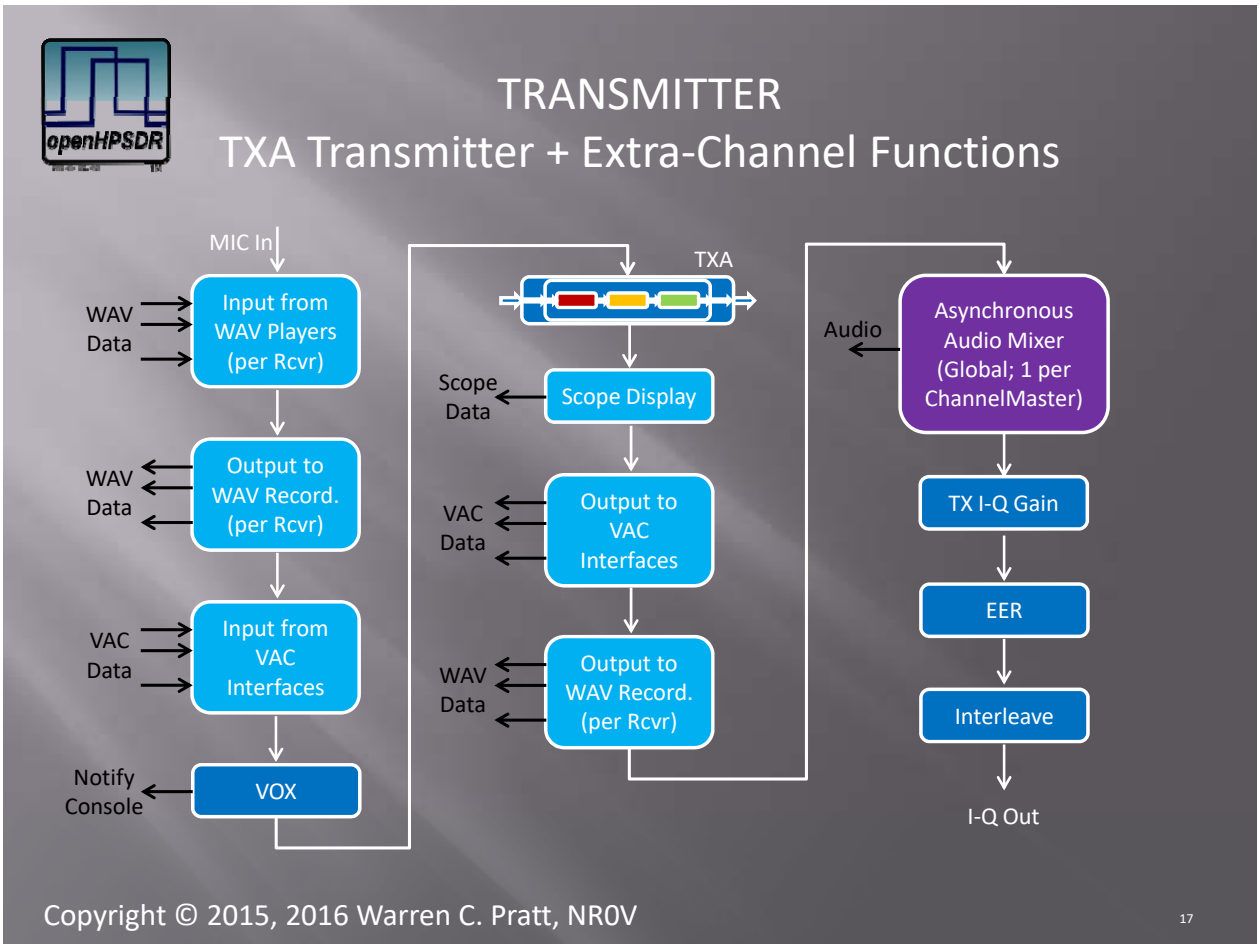
Functional blocks in the TRANSMITTER diagram that are shown in dark-blue are those that are instantiated/owned by the transmitter. Those shown in light-blue are instantiated/owned by RXn Receivers; however, they are accessed by the transmitter code as transmit data is processed. For example, each receiver has a WAV Recorder. If a receiver's WAV Recorder is recording, then, during receive it gets data from the appropriate receiver and during transmit it gets data from the transmitter.

VOX: One per transmitter. Owned/instantiated by the transmitter. Detects audio above VOX threshold and informs the console code. Code is found in files "vox.c" and "vox.h".

TX I-Q Gain: One per transmitter. Owned/instantiated by the transmitter. There are two occasions where the TX I and Q values must be multiplied by a gain factor: (1) for the Penelope transmitter card, and (2) for the "ALC" external amplifier protection feature. Code is found in files "txgain.c" and "txgain.h".

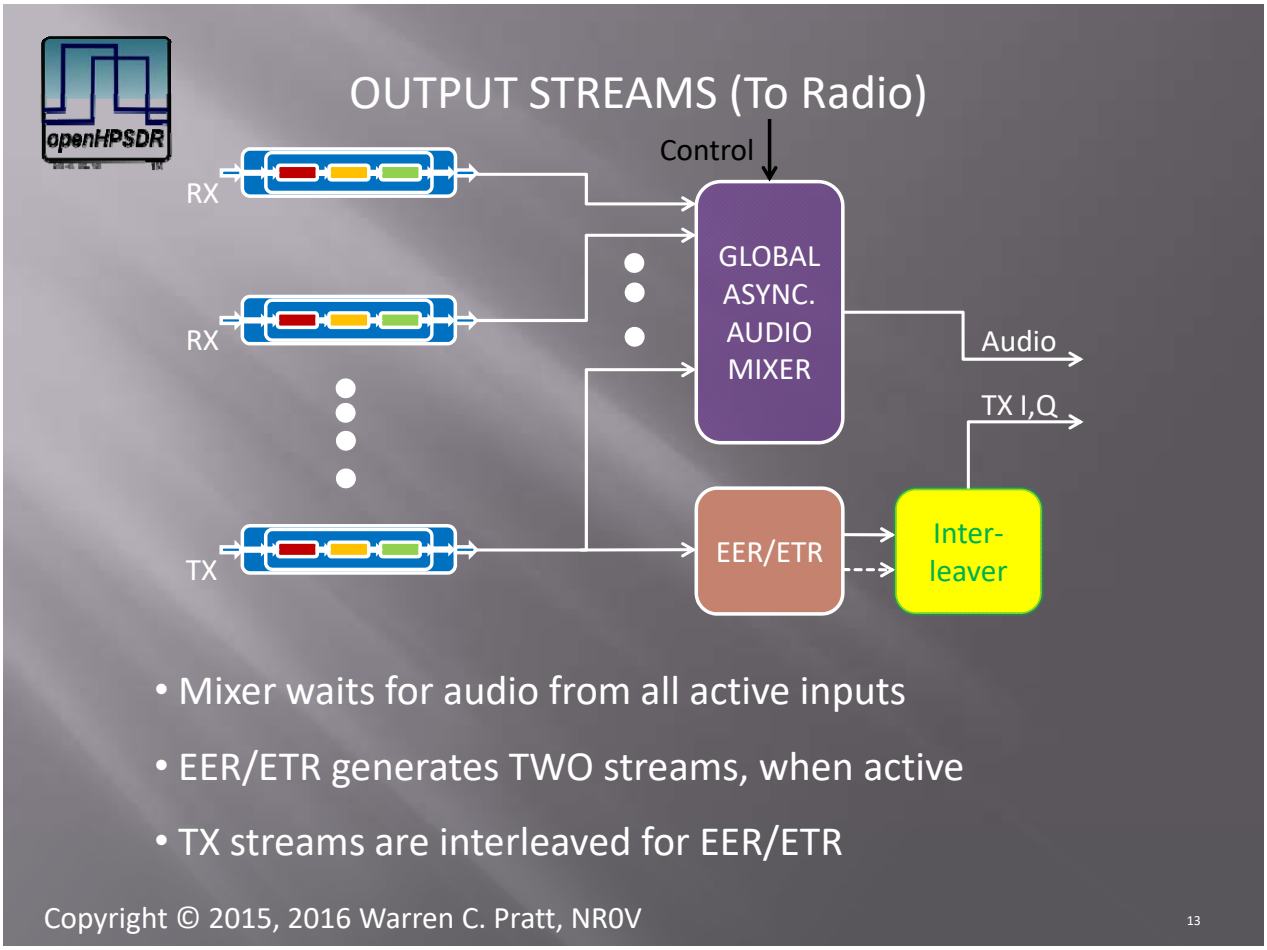
EER: One per transmitter. Owned/instantiated by the transmitter. Separates the transmit I-Q signal into multiple streams as needed for Envelope Elimination and Restorations and Envelope Tracking modes of transmission. Signal processing code is part of WDSP; however, a control interface is found the file "zeer.c".

Interleave: One per transmitter. Owned/instantiated by the transmitter. Used to interleave multiple data streams into a single interleaved stream. Used with EER. Code is found in the files "ilv.c" and "ilv.h".

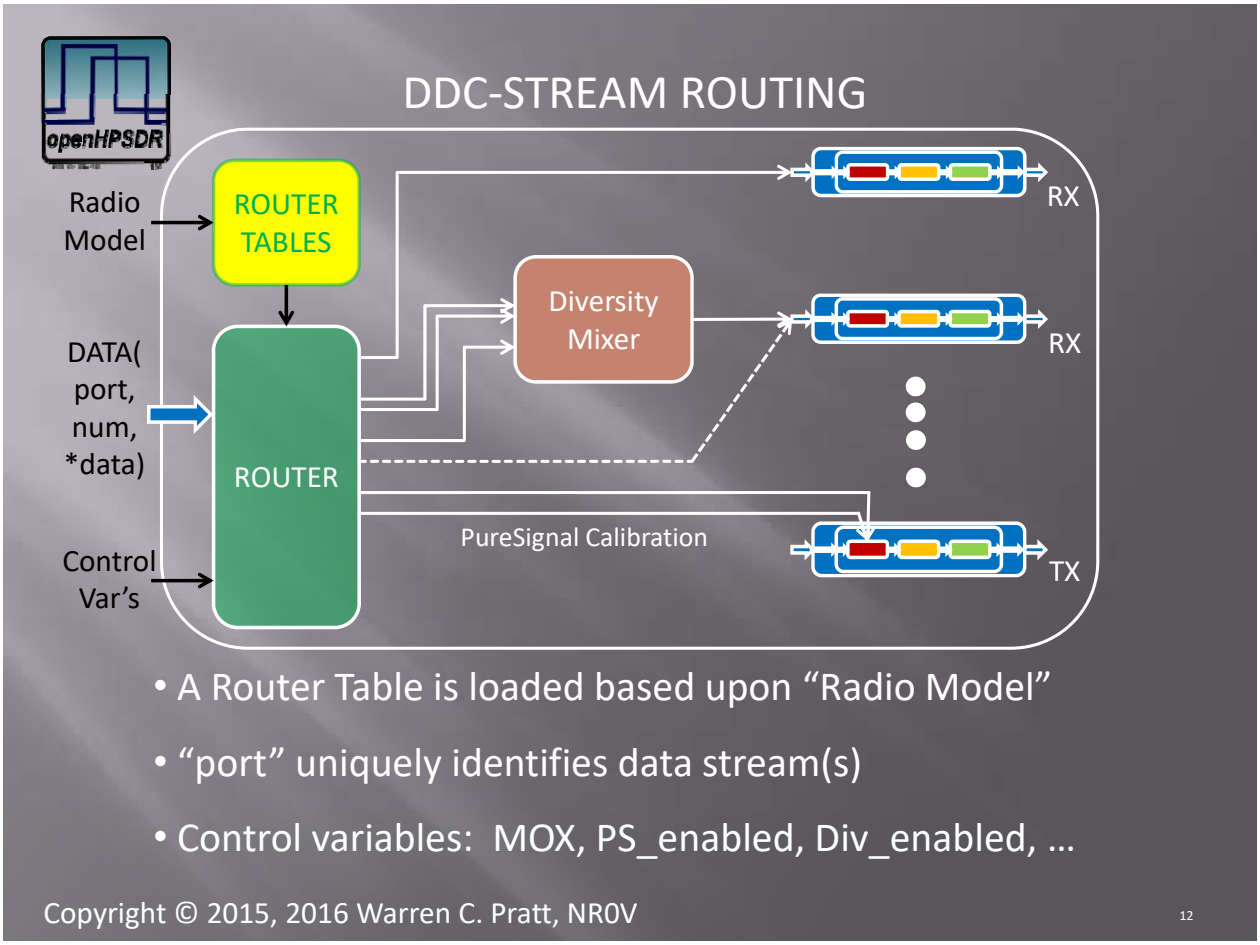


Summary of Outbound Data (To Radio)

There are two types of outbound data going back to the radio hardware: Audio samples, and TX I-Q samples. The following diagram provides a simplified summary of how those are derived. Note that, along with all RX outputs, the TX output also goes to the Audio Mixer. This is for the transmit Monitor function.



The Router



Setting Up The Router

The most difficult part of setting up the router is deciding what you want it to do. It is suggested to prepare a spreadsheet (example below) containing an exhaustive display of all cases. Once this is prepared, it is a relatively easy mechanical process to, from the spreadsheet, create the set of numbers that will go into the router table.

There are two ways to control the data-flow through the router (i.e., to control what input goes to what output(s)): (1) by the router table that is loaded, and (2) by designating and setting router control variables. It is recommended that a new router table be loaded only when there is a major configuration change in the radio, for example, when the hardware model is switched from one with two DDCs to one with four DDCs. "Variables" can be used to control the routing within the specifications of the table. For example, in Thetis, the routing changes depending upon (1) the state of MOX, (2) whether Diversity Reception is enabled, and (3) whether PureSignal is enabled. So, in Thetis, there are three router control variables used: MOX, Diversity, and PureSignal. This means that there are 2^3 or 8 possible combinations of those variables that must be considered. For any given router

table, we must decide, for each DDC, where its data stream is to go for each of the eight combinations of the router control variables.

Consider the following example spreadsheet for Thetis:

Model	Firmware DDC	Inbound() and InboundBlock() Calls							
		!MOX && !Diversity && !PureSignal	!MOX && !Diversity && PureSignal	!MOX && Diversity && !PureSignal	!MOX && Diversity && PureSignal	MOX && !Diversity && !PureSignal	MOX && !Diversity && PureSignal	MOX && Diversity && !PureSignal	MOX && Diversity && PureSignal
(Angelia || Orion):	DDC0*			IBB(0)	IBB(0)		IBB(1)	IBB(0)	IBB(1)
	DDC1*								
	DDC2	IB(0)	IB(0)			IB(0)	IB(0)		IB(0)
	DDC3	IB(1)	IB(1)	IB(1)	IB(1)	IB(1)	IB(1)	IB(1)	IB(1)
(Hermes || Hermes-E):	DDC0**	IB(0)	IB(0)	IBB(0)	IBB(0)	IB(0)	IBB(1), IBB(3)	IBB(0)	IBB(1), IBB(3)
	DDC1**	IB(1)	IB(1)			IB(1)			
* DDC0 and DDC1 are always synchronous, output on DDC0, port 1035.									
** DDC0 and DDC1 are synchronous at some times and not at others.									
NOTE: An IBB() call implies that the receivers are operating synchronously.									

First of all, there are TWO complete router tables displayed here, one for four DDC radios (Angelia and Orion) and one for two DDC radios (Hermes and Hermes-E). Then, for each of those two router tables, we must consider what happens to the output of each DDC.

Across the top of the chart, we see the eight combinations of the three control variables. We must determine the destination of each DDC output for each combination ... hence, the cells of the spreadsheet.

In some cases (see footnotes on the spreadsheet) DDC0 and DDC1 will be operating "synchronously" and interleaved data is being received from the pair of DDCs. In those cases, two cells of the spreadsheet have been merged and there is an 'IBB(y)' designation in such a cell. When DDCs are NOT operating synchronously and data from a DDC is being returned alone in its own packets, there is an 'IB(x)' designation in the corresponding cells. 'IB(x)' and 'IBB(y)' stand for the 'Inbound(x)' and 'InboundBlock(y)' calls that the router is to make. These were discussed above in the section "Following The Data Through The Code - DDC Samples."

The 'x' values in an 'IB(x)' call specify a ChannelMaster 'stream'/'inid' identifier as discussed in the section on "Identifiers" under "Instantiatiating the ChannelMaster."

The 'y' values in an 'IBB(y)' call designate the destination for samples sent to InboundBlock(y). There are currently four possible cases, also discussed above in "Following The Data Through The Code - DDC Samples." Briefly reiterating:

0 - data goes to diversity mixer

1 - data goes to PureSignal calibration function

2 - data goes to software receivers of rxid=0 and rxid=1

3 - data from DDC0 goes to software receiver rxid=0 and data from DDC1 is discarded.

Now, let's work through the table entries for Hermes and Hermes-E, just as examples.

- In the first two columns, we're in receive mode (!MOX) and Diversity is not enabled. The two DDCs are set to operate asynchronously (which implies that they can have different sample rates) and the data from each is sent to a respective software receiver -- DDC0 goes to receiver rxid=0 and DDC1 goes to receiver rxid=1.
- In the third and fourth columns, we're still in receive (!MOX) but now Diversity is enabled. For Diversity to function, the receivers must be set up to operate synchronously. Hence, we must use an IBB call with the destination of the diversity mixer -- the call is 'IBB(0)'.
- In the fifth column, we're transmitting (MOX) but neither Diversity nor PureSignal are in use. The DDC data is sent to the two software receivers, just as in columns one and two.
- In the sixth column, we're transmitting with PureSignal enabled and Diversity not enabled. For PureSignal calibration, the two DDCs must operate synchronously. Hence, we have the call 'IBB(1)', i.e., synchronous operation with the data going to the PureSignal calibration function. We would also like to see the PureSignal results on the panadapter using the DUP functionality. Therefore, we make a second call to 'IBB(3)' routing the results to the first software receiver, rxid=0. Note that the frequency scale on the panadapter must be changed to match the transmit frequency because both receivers must be on the transmit frequency for PureSignal calibration. The transmit frequency may or may not be the same as the receive frequency.
- In the seventh column, we're transmitting with Diversity enabled (PureSignal not enabled). The receivers must be synchronous for Diversity and data goes to the diversity mixer, i.e., 'IBB(0)'.
- In the eighth column, we're transmitting with both Diversity and PureSignal enabled. The receivers must be synchronous. For PureSignal, the two DDCs must be on the transmit frequency; however, for Diversity, the two DDCs would be on the receive frequency which may be different. Since we're transmitting, we deemed PureSignal more important; however, because we'd like to monitor PureSignal performance in DUP mode on the panadapter, we'll route the signals to BOTH the first software receiver and to the PureSignal calibration function using two calls, 'IBB(1)' and 'IBB(3)'. Note that the frequency scale on the panadapter must be changed to match the transmit frequency because both receivers must be on the transmit frequency for PureSignal calibration.

IMPORTANT NOTE: The router table ONLY controls the routing of data. However, other control functions must correspond to what the router is doing as expressed in the router table. For example, in some situations the two DDCs must be set to run synchronously and at other times not. This must be controlled in other code; although, the above spreadsheet clearly shows the required settings! Other settings which must be controlled are (1) whether DDCs are enabled, (2) the DDC frequencies which may change depending upon variables like MOX, and (3) the various DDC sample rates.

Having our spreadsheet prepared, it's now time to actually set up the router. To do so, we use the following call:

```
void LoadRouterAll (
void* ptr,              // pointer to router data structure.  We will use the following 'id'
                        // instead; so, set this to 0.
int id,                 // 'id' of the router.  There's only one in the ChannelMaster at this point;
                        // so, set this to 0.
int ports,              // maximum number of network ports on which data will be received
int calls,              // maximum number of function calls the router is to make to 'IB' or
                        // 'IBB' for each port
int varvals,            // number of combinations of router control variables; e.g., for 3
                        // variables, this is 2^3 = 8
int* nstreams,          // array containing the maximum number of streams from each port
int* function,          // array showing functions to be called:  0 = none, 1 = 'IB', 2 = 'IBB'
int* callid);           // array showing parameter to be passed in each call
```

Continuing with the Hermes example from above, the parameters would be as shown below. Note that for '*function ' and '*callid ' the eight columns correspond to the columns of the spreadsheet.

```
ptr = 0;
id = 0;
ports = 2;              // we use port 1035 for DDC0 data and port 1036 for DDC1
calls = 2;              // in some cases, we must make two 'IBB' calls for a single set of data
varvals = 8;            // eight combinations of the three router control variables
*nstreams = {2, 1};     // two (interleaved) streams from DDC0 and one stream from DDC1
*function = {   1, 1, 2, 2, 1, 2, 2, 2,     // DDC0, port 1035, First Call (0 = none, 1 = 'IB', 2 = 'IBB')
                0, 0, 0, 0, 0, 2, 0, 2,     // DDC0, port 1035, Second Call
```



```
                1, 1, 0, 0, 1, 0, 0, 0,     // DDC1, port 1036, First Call
                0, 0, 0, 0, 0, 0, 0, 0};    // DDC1, port 1036, Second Call [there are none]
*callid = {     0, 0, 0, 0, 0, 1, 0, 1,     // DDC0, port 1035, First Call
                0, 0, 0, 0, 0, 3, 0, 3,     // DDC0, port 1035, Second Call
                1, 1, 1, 1, 1, 1, 1, 1,     // DDC1, port 1036, First Call (if no call, value doesn't matter)
                1, 1, 1, 1, 1, 1, 1, 1};    // DDC1, port 1036, Second Call
```

After the router table is loaded, we still need a call to set the value of the router control variables. That's done as follows:

```
void LoadRouterControlBit (void* ptr, int id, int var_number, int bit);
```

where 'ptr' and 'id' can be set to 0, as in the LoadRouterAll(...) call. 'var_number' specifies which variable is being set and 'bit' specifies the value of the variable where 0 is 'false' and 1 is 'true'. Relating to the example spreadsheet, 'var_number' 0 is "PureSignal", 'var_number' 1 is "Diversity", and 'var_number' 2 is "MOX".

The Asynchronous Audio Mixer

Combining multiple input streams of sampled audio into a single audio stream can be as simple as adding together the respective samples of the various input streams. However, the new ethernet protocol provides no guarantees of the arrival times of the raw data (DDC data or MIC samples) that will be processed to produce the audio input streams. There are not even any guarantees about the order and order-consistency of various data streams. Additionally, there is variability in the processing times of various data streams to produce audio because this processing occurs per the operating system scheduling of individual threads. For these reasons, a mixer is needed that can synchronize the various input audio streams and then add the respective samples together. The Asynchronous Audio Mixer [code in aamix.c and aamix.h] does this.

Additionally, all input streams are not necessarily at the same sample rate. This mixer also converts all incoming streams to the output sample rate before summing them.

The Global Mixer, the one creating the audio stream that is sent back to the radio hardware, is instantiated during instantiation of the ChannelMaster. Sample rates are automatically set based upon ChannelMaster settings at instantiation. There are, however, some interactions required from the console and these are discussed now.

To synchronize the various input audio streams, the mixer waits until it has enough samples of each input stream to create an output buffer, then it mixes and creates the output buffer. It follows that if some input stream is NOT providing any input, the mixer will block indefinitely waiting for input. This could happen, for example, if the mixer is expecting audio from a particular software receiver but the DDC feeding the receiver has been turned OFF. Therefore, the mixer must be told, of the possible input streams, which ones are currently active and which are not. Two alternative calls are provided for doing this.

The first sets the state (active/inactive) of a single stream.

```
void SetAAudioMixState (void* ptr, int id, int stream, int state);
```

The second sets the state of multiple streams using a single call.

```
void SetAAudioMixStates (void* ptr, int id, int streams, int states);
```

In some cases, even though data for a particular stream is known to be flowing, it is desirable to turn OFF having it mixed into the output. The mixing state of a particular stream can be set using:

```
void SetAAudioMixWhat (void* ptr, int id, int stream, int state);
```

Volume settings are also available -- both an overall output volume and volume settings for individual streams.

```
void SetAAudioMixVolume (void* ptr, int id, double volume);

void SetAAudioMixVol (void* ptr, int id, int stream, double vol);
```


Instantiating the ChannelMaster

The ChannelMaster is instantiated and default values are set for operation using a sequence of calls. The first call to be made is SetRadioStructure(...). This call sets basic parameters that will be used to instantiate sub-units and allocate storage.

For this call, you must select the MAXIMUM number of streams, receivers, transmitters, sub-receivers, and specials that will ever be used. Not all these need be active in operation; however, resources for them must be allocated at this point.

```
void SetRadioStructure (
    int nstreams,           // total number of input data streams, including DDC, MIC, and others
                            // for special units
    int nrcvrs,             // number of software receivers
    int nxmtrs,             // number of software transmitters
    int nsubrx,             // number of sub-receivers per software receiver (main+one_sub = 2)
    int nspc,               // number of TYPES of non-rx/non-tx special units, for example for a
                            // "stitched" panadapter display
    int* spc,               // array giving number of special units OF EACH TYPE
    int* MAXInBound,        // array giving max number of samples in a call to Inbound(), per stream
                            // (determined by number of samples per packet)
    int MAXInRate,          // maximum sample rate of any input data stream
    int MAXAudioRate,       // maximum audio output rate of any DSP channel (include receivers
                            // and transmitter monitor audio)
    int MAXTxOutRate);      // maximum transmitter channel output sample rate
```

Next, we send function/method pointers for certain calls that the ChannelMaster will need to make to the console code. It is recommended that these calls be consolidated into one simple function. In the case of the ChannelMaster as used in Thetis, these function/method pointers comprise:

- Pointer to the VOX function. This is needed as the C# console code must be informed that MOX is to be 'true' or 'false'.
- Pointers to the Create_Wave_Player and Create_Wave_Recorder functions. These are needed since the Wave_Player and Wave_Recorder codes are in C# in the Thetis console code. I.e., they have not been translated to C.
- Pointer to the Create_Scope function. This is needed since the Scope code is in C# in the Thetis console code. I.e., it has not been translated to C.

The specific calls are:


```
void SendCBPushVox (int xmtr_id, void (__stdcall *pushvox)(int id, int active));

void SendCBCreateWPlay (void (__stdcall *create_WavePlay)(int id));

void SendCBCreateWRecord (void (__stdcall *create_WaveRecord)(int id));

void SendCBCreateScope (void (__stdcall *create_Scope)(int id));
```

Next, we set the default sample rates. These rates can be changed later; however, in some cases, it may be desirable just to always leave them at their default values.

```
void set_cmdefault_rates (

        int* xcm_inrates,       // sample rates for each of the input data streams, ordered as DDC
                                // streams, MIC stream, Specials streams

        int aud_outrate,        // ChannelMaster audio output sample rate

        int* rcvr_ch_outrates,  // output rates of all RXA WDSP channels

        int* xmtr_ch_outrates); // output rates of all TXA WDSP channels
```

At this point, we are ready to make the very important calls that use all the previously entered information to complete instantiation of the ChannelMaster:

```
void CreateRadio();             // creates functional blocks

void create_rnet();             // creates network layer
```

The ChannelMaster is now instantiated and ready for use. Optionally, at this point, we may wish to set a few parameters that (1) are to be different than default values, and/or (2) will be constants, i.e., their values will NOT be changed during the operation of the program. Since they are constant, we may as well do that here and not bother to do it elsewhere. To use some of these calls, you will need "Identifiers". For example, if you want to set a parameter for the TXA in WDSP, you need to know the Channel number. "Identifiers" are covered in the section of the same name, below.

The firmware interpolation filters are designed differently for the new Ethernet protocol and the old USB protocol. As a result, for Thetis, we need to turn on a special "CFIR" filter in WDSP that is not used in PowerSDR. It is OFF by default. The call to do so is:

```
void SetTXACFIRRun (int channel, 1);
```

There are a few constants we can set for PureSignal. All current radio models are set up to use Stream0 for off-air RX feedback and to use Stream1 for loopback TX feedback. We can specify those streams. We can also set the PureSignal Feedback rate and set the peak level delivered by the new Ethernet protocol firmware.

```
void SetPSRxIdx (int txid, int stream);         // Thetis uses:  SetPSRxIdx (0, 0);
```



```
void SetPSTxIdx (int txid, int stream);              // Thetis uses SetPSTxIdx (0, 1);

void SetPSFeedbackRate (int channel, int rate); // Thetis uses SetPSFeedbackRate (channel, 192000);

void SetPSHWPeak (int channel, double peak);  // Current NP Firmware:  peak = 0.2899
```

The transmit panadapter/waterfall generation is handled differently in Thetis compared to PowerSDR; so, there are a couple things that need to be set differently from WDSP defaults for that also:

```
void TXASetSipMode (txchannel, 1);

void TXASetSipDisplay (txchannel, txid);
```

Freeing Resources

When the application is to be closed, ChannelMaster resources should be freed. This is done with the simple call:

```
void DestroyRadio();
```

Identifiers & Related Utilities

There are several identifiers and related values that are needed to address various items. There are also utilities in the ChannelMaster that, given some information, can return to you the identifier that you need.

First of all, we need to understand the variable 'stype', the "stream_type". Several types of data streams enter the ChannelMaster for processing: DDC (receiver) streams, MIC (transmitter) streams, and data streams for 'special' units. Each of these types is assigned an 'stype'.

0 - receiver streams

1 - transmitter streams

2 - special stream (more of these could be defined as 3, 4, ... n)

"Streams" are also assigned an identifier, beginning with 0 and going up to the total number of input streams to the ChannelMaster. This identifier is named 'stream' or also 'inid'.

Software Receivers are assigned identifiers, beginning with 0 and going up to the total number of software receivers. This identifier is named 'rxid'.

Software Transmitters are assigned identifiers, beginning with 0 and going up to the total number of software transmitters. This identifier is named 'txid'.

Special Units are assigned separate identifiers for each type of special unit. These also begin with 0 and go up to the total number of units of the type. For the first special type, this identifier is named 'sp0id'.

As discussed in the "WDSP Guide", each WDSP channel is assigned a channel number. This identifier is called 'channel' or also 'chid'.

Given these definitions, the following calls can be used to return the identifiers that are needed for any specific call to WDSP or to the ChannelMaster:

```
int stype (int stream);

int rxid (int stream);

int txid (int stream);

int sp0id (int stream);

int chid (int stream, int subrx);          // TXA has no 'subrx'; set to 0.

int inid (int stype, int id);              // 'id' is one of 'rxid', 'txid', 'sp0id'
```

There are other related utility calls that, using these identifiers, can be used to retrieve information about particular parameters.

```
int getInputRate (int stype, int id);          // returns the sample rate of an input stream

int getChannelOutputRate (int stype, int id);  // returns the output rate of a WDSP Channel

int getbuffsize (int rate);                    // returns a buffer size being used in xcmaster()
```

Changing Sample Rates

As mentioned in the section on "Instantiating the ChannelMaster," the default sample rates that are specified when instantiating can be later changed during operation.

To change the input rate that the ChannelMaster is expecting from radio hardware, use the following call. Note that, since the WAV Player and WAV Recorder are currently in C# code, it is left to the console to change the expected rates for them separately. The display samplerate is currently also set in the separately in the console code.

```
void SetXcmInrate (int in_id, int rate);
```

The sample rate of ChannelMaster audio output going back to the radio hardware is changed with the the following call. Note that this will usually NOT change and will remain at the default rate set at instantiation.

```
void SetCMAudioOutrate (int id, int rate);
```

The output sample rate of a particular software receiver can be set using the following call. However, this call is not normally used as values normally stay at their defaults. Note that this does not set the

Scope, WAV Recorder, and WAV Player since that code is currently in C#, i.e., this must be done separately by the console.

void SetRcvrChannelOutrate (int rxid, int rate, int state);

The Transmitter output sample rate is set using the following call. Note that the transmitter display sample rate, the Scope rate, and the WAV Recorder rate must be set separately by the console.

void SetXmtrChannelOutrate (int txid, int rate, int state);

Command & Control

Computer to Radio Hardware

The console makes various calls to set parameters in a data structure. When the console changes a parameter value, this triggers sending a new packet containing the modified value. In [network.c], are found functions CmdGeneral(), CmdHighPriority(), CmdRx(), and CmdTx(), corresponding to the various types of computer-to-radio packets.

Radio Hardware to Computer

The High-Priority Status Packet is unpacked in ReadThreadMainLoop() [network.c] and its values are stored. Calls are provided such that the console can poll the various values contained in this packet.

