Commentary on DttSP

By Jonathan Naylor ON/G4KLX

The DttSP DLL or executable is the heart of the SDR1000 and other software defined radios. It provides transmit and receive routines for almost all of the amateur radio modes including CW, SSB, AM and FM. Unfortunately it is not well documented and using it requires reading through the source code for PowerSDR, DttSP-Shell and the Java GUI for the Softrock.

DttSP currently exists as a DLL under Windows or as a standalone executable under Linux, in the latter case data and commands are sent to and from DttSP via pipes.

This document attempts to explain the operation of the DLL version of DttSP shipped with PowerSDR 1.6.0 and is probably applicable to the Linux version also. It has been stated that the internals of DttSP will be refactored, but I am sure that the basic processing blocks will not change appreciably.

General

Starting DttSP

Before starting to use DttSP the function `Setup_SDR()` in `update.c` should be called to initialise variables and structures. To close DttSP and release its resources the `Destroy_SDR()` function in `update.c` should be called.

DttSP runs as a separate thread which is created by the host program. This thread should call `process_samples_thread()` function in `winmain.c` which will then block on a semaphore (`top.sync.buf.sem`) until data arrives. When data arrives the thread is unblocked and data is transferred from the input and output ring buffers to the input and output ring buffers (`top.hold.buf.l` and `top.hold.buf.r`) that are used by the transmit and receive routines, this is done in `gethold()` in `winmain.c`. The same ring buffers are used for input and output.

After checking that the update semaphore (`top.sync.upd.sem`) is not being held by another thread updating one of the important internal variables. This top level thread runs until the running variable (`top.running`) is set to FALSE, this is done within the `Destroy_SDR()` function.

Provided that all is fine, the state variable (`top.state`) is checked to determine the top level mode of DttSP is. In the `RUN_MUTE` mode, the input data is ignored and only empty buffers containing zeroes will be made available to the calling application. In the `RUN_PASS` mode nothing is done, in the `RUN_SWCH` mode the input data is ramped from zero to their full values before passing them to `process_samples()` in `sdr.c`. In the `RUN_PLAY` mode the input data is passed straight to `process_samples()`. The default is `RUN_PLAY`. The process samples thread keeps looping until no more data is available, checked in `canhold()` in `winmain.c` before blocking again on the main data semaphore.

Within `process_samples()` the decision is made whether the data is to be used for the transmitter or received based on the setting of `uni.mode.trx`. This is set via the DLL function `SetTRX()` in `update.c` or the pipe command “`setTRX`” aith the value of 0 for receive and 1 for transmit. In the DLL function much more care is taken to set the values of the variables ready for the use of the `RUN_SWCH` mode which is also set in this function. It also includes what appears to be a hard coded sample rate of 48000 Hz too.

If the chosen mode is receive (`RX`) then for each defined received, up to a maximum of four, the data is duplicated and passed to `do_rx()` in `sdr.c`, passing the receiver number as an argument if that receiver is active (`uni.multirx.act`). The output audio is then multiplied if

the late receive mixing flag is set (uni.mix.rx.flag) by the value uni.mix.rx.gain, these are set by the pipe commands of "setAuxMixSt" and "setAuxMixGain", there is no DLL call equivalent of these commands. The choice of which of these receivers to use is set by the pipe command "setRXListen", it has no DLL call equivalent. The receiver activation flag is set by the "setRXOn" and "setRXOff" pipe commands, yet again no DLL call equivalent exists limiting the DLL version of DttSP to one receiver.

On transmit (TX), things are easier since only one transmitter is allowed. The input audio is multiplied if the early mixing transmit flag is set (uni.mix.tx.flag) with the value uni.mix.tx.gain. Like the receiver path these are set by the "setAuxMixSt" and "setAuxMixGain" pipe commands. The transmit chain is then called through the do_tx() function in sdr.c.

The Signal Input/Output

The data is passed to and from DttSP via any of three functions, Audio_Callback(), Audio_CallbackIL() and Audio_Callback4IL() all in winmain.c. These essentially perform the same task of taking the input data, placing it into the input I and Q ring buffers (top.jack.auxr.i.l, top.jack.auxr.i.r, top.jack.ring.i.l, top.jack.ring.i.r), getting any output data from the output I and Q ring buffers (top.jack.auxr.o.l, top.jack.auxr.o.r, top.jack.ring.o.l, top.jack.ring.o.r), and finally setting the run semaphore top.sync.buf.sem. In the event of a ring buffer underrun or overrun the monitor thread is unblocked via a semaphore (top.sync.mon.sem) which prints an error message to stderr. Unless redirected this will disappear on Windows systems and the errors will not be logged. The use of four buffers instead of two for both input and output is confusing, this will be investigated further.

The differences between the different audio callback functions is to do with the format of the data. With Audio_Callback() the data is assumed to be held in separate float arrays, one each for the I and Q channels and separate for transmit and receive, four buffers in total. The number of samples for all is passed also. Audio_CallbackIL() is well suited for use with the input and output of audio cards and files which often provide the data in an interleaved format, with this function only one float array is needed for each direction leading to the use of two buffers. AudioCallback4IL() is similar to Audio_CallbackIL() in only using two buffers, but the data from the top.jack.ring and top.jack.auxr ring buffers are interleaved fully, unlike in Audio_CallbackIL() where they appear to overwrite each other.

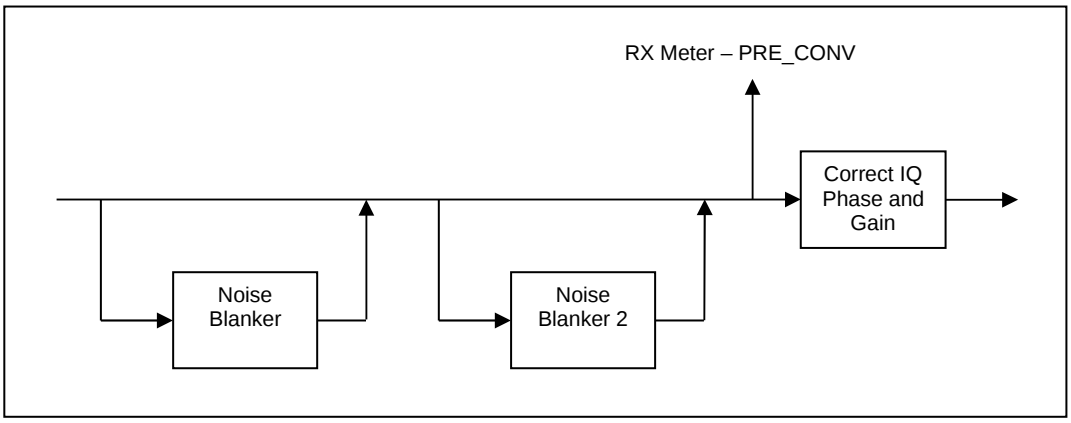
An oddity of DttSP is that in the interleaved data buffers, the order of the samples is "Q1 I1 Q2 I2 …. Qn In", which is opposite to what you would expect. Getting the order wrong causes effects such as sidebands being swapped!

All of these functions are well suited to use in calls from another thread as they never block.

The Receive Chain

The process within the DttSP receiver is split into three parts, a pre-processing section that is common to all modes, the actual demodulator and finally a mode independent post processing section.

The pre-processing is performed in do_rx_pre() in sdr.c which is passed the receiver number as an argument.



The Noise Blankers

The first stages of the receive chain are the two noise blankers. These operate on the full bandwidth of the receiver and operate on different principles and offer different noise blanking performance. The first noise blanker is optimised for reducing the effects of impulse noise, for example from badly suppressed car engines. This is enabled or disabled via the DLL function `SetNB()` or the pipe command "`setNB`". The threshold for the noise blanker is set by the DLL function `SetNBvals()` and the pipe command "`setNBvals`" and they take a floating point number between 1.0 and 20.0, the default is 3.3. . The code for this noise blanker is `noiseblanker()` in `noiseblanker.c`.

Noise Blanker 2 is called a mean noise blanker and is presumably used to blank other forms of noise. Its is enabled via the DLL function `SetSDROM()` and the pipe command "`setSDROM`". The threshold is set via the DLL function `SetSDROMvals()` and the pipe command "`setSDROMvals`" and is a floating point number between 0.0 and 15.0, the default is 2.5. The code for this noise blanker is `SDROMnoiseblanker()` in `noiseblanker.c`.

At this stage the values of the I and Q channels are tapped off to the function `do_rx_meter()` in `sdr.c` and stored for potential later use. These are converted into decibels. These are available as meter types of `ADC_REAL` and `ADC_IMAG`.

The meter values are available via the DLL function `Calculate_Meters()` in `update.c`. This takes the meter type as an argument and returns a float which is the value of the meter. It is not possible to query the transmit meters when receiving and vice versa.

I/Q Balancing

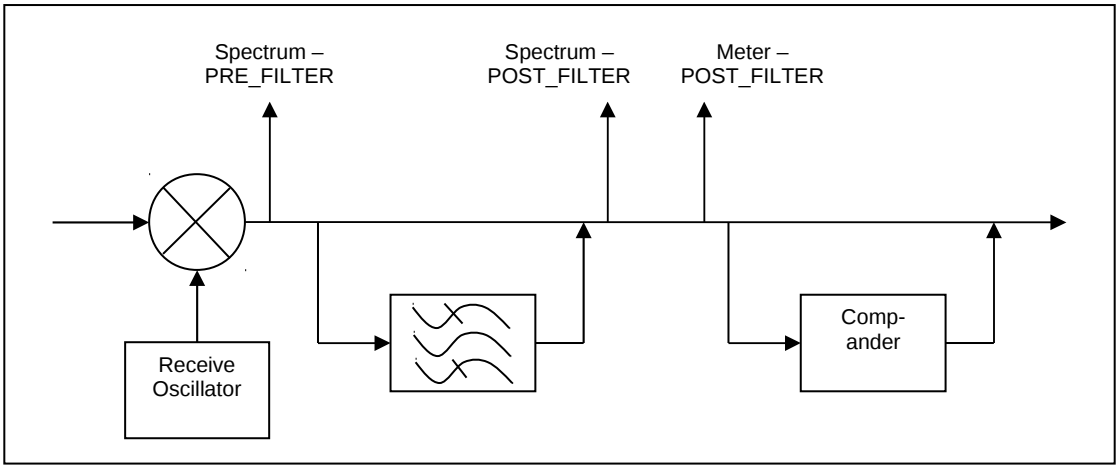
The imperfections within the hardware of the receiver leads to an imbalance of the I and Q channels, this takes the form of imbalances in amplitude and phase. For optimum performance these need to be balanced out. This is done by the function `correctIQ()`in `correctIQ.c`. The imaginary (Q) part of the signal is multiplied by the a proportion of the real (I) part of the signal based on the value of the phase correction, the real (I) part is then scaled depending on the value of the gain correction. The default phase correction is 0.0 and the default gain correction is 1.0.

The setting of the correct factors can be done via a number of DLL functions and pipe commands. They all operate on the same internal variables:

DLL Function	Pipe Command	Arguments	Comments
`SetCorrectIQ`	`setcorrectIQ`	Phase – float Gain - float	Phase = 0.001 * value, Gain = 1.0 + 0.001 * value
`SetCorrectIQPhase`	`setcorrectIQphase`	Phase - float	Phase = 0.001 * value
`SetCorrectIQGain`	`setcorrectIQgain`	Gain - float	Gain = 1.0 + 0.001 *

			value

The values passed to DttSP are between –400 and 400 for the phase and between –500 and 500 for the gain. These map onto values for the phase of between –0.4 and +0.4 and for the gain of between 0.5 and 1.5.



Oscillator and Mixer

If the receive oscillator is set to zero then the frequency mixing step is bypassed. The receive oscillator is set by the DLL function `SetOsc()` and the pipe command "`setOsc`" with the first argument of the frequency and the second argument set to zero to indicate that it is the receive oscillator that is being set. The frequency cannot be more than half the sampling frequency, if it then the command is rejected. The oscillator is the function `ComplexOsc()` in the file `oscillator.c`. The actual mixing is done by a complex multiply function.

Spectrum Processing

At this stage the signal is sampled for potential use in a spectrum display. This spectrum data is available as pre-filter data.

The type of spectrum display wanted is set by the DLL function `SetPWSmode()` and the pipe command "`setSpectrumType`". The operation of these two entry points is different:

The values used by `SetPWSmode()` do not match those of the pipe command, even though it uses the pipe command itself (as do many of the DLL functions).The mapping is as follows:

DLL Function	Pipe Command	Meaning
`SetPWSmode(0)`	`setSpectrumType 2`	Post filter spectrum
`SetPWSmode(1)`	`setSpectrumType 1`	Pre filter spectrum
`SetPWSmode(2)`	`setSpectrumType 0`	Semi raw (unused)

The pipe command can take a variable number of arguments, the first is the type of spectrum display required, the second is the scale, and the third is the receiver number. The default scale is `SPEC_PWR` which is a decibel scale, the alternative is `SPEC_MAG` which is raw values.

The system default is a post filter with a decibel scale.

There are a number of DLL functions that return the spectrum. The function `Process_Spectrum()` in `update.c` returns a 4096 point FFT of the post filter spectrum. Calling this function also sets the type of spectrum to collect, the same as calling `SetPWSmode()`. This means that the first time that this function is called, if a different type of spectrum has already been selected then the wrong spectrum will be returned, subsequent calls will be correct. The function `Process_Panadpter()` also in update.c also returns a

4096 point FFT, but that is the pre filter spectrum, the same general comments apply to this function as to `Process_Spectrum()`.

The DLL functions `Process_Phase()` and `Process_Scope()` in update.c operate on the spectrum which is collected after the AGC (see below), they both return a spectrum from an FFT but return different data. `Process_Phase()` returns both the I and Q output of the FFT interleaved into a buffer, `Process_Scope()` is similar but only returns the I output of the FFT.

In order to improve the resolution of the FFT, it is possible to enable the use of a windowing function. The wanted window function is selected via the DLL function `SetWindow()` in `update.c` and the pipe command "`setSpectrumWindow`". This calls the `makewindow()` function in `window.c` to create the array of float which are the window function.

The values of the different window types are:

Number	Window
0	Rectangular (i.e. no window)
1	Hann
2	Welch
3	Parzan
4	Bartlett
5	Hamming
6	Blackman 2
7	Blackman 3
8	Blackman 4
9	Exponential
10	Riemann
11	Blackman Harris
12	Nuttall

The Main Filter

If the mode is other than SPEC, then the next stage is the filter. The filter is the function `filter_OvSv()` in file `ovsv.c` which implements a FIR bandpass filter. Being a FIR means that the phase of the signal is preserved which is ideal for all modes despite not necessairily having the same level of performance as an IIR. The OVSV stands for Overlap Value Set Value, which is an efficient implementation of an FIR using convolution with a FFT.

A new filter setting is created by calling `newFIR_Bandpass_COMPLEX()` in `filter.c` which initially creates a Blackman Harris window which is then scaled by the filter parameters. The co-efficients are then passed to `newFiltOvSv()` where it is convolved for use in the filter.

The entry points into DttSP for setting the receive filter is the DLL function `SetFilter()` and the pipe command "`setFilter`". The is used for setting both the transmit and receive filters, the arguments are the low frequency, high frequency and whether it is for transmit or receive. The frequencies are in Hertz and are checked for validity before being used. With the pipe command, the transmit/receive item is optional and its absence indicates that the receive filter is being set.

The frequencies of the filter are centred around 0 Hz where is the nominal carrier frequency. This means that to pass a lower sideband signal, the frequencies should be negative, the values of the pre-set filters within the SDR-1000 console are:

Bandwidth	Mode	Low Freq (Hz)	High Freq (Hz)
10 kHz	AM/FM	-5000	5000
	USB	100	10100
	LSB	-10100	-100

6 kHz	AM/FM	-3000	3000
	USB	100	6100
	LSB	-6100	-100
4 kHz	AM/FM	-2000	2000
	USB	100	4100
	LSB	-4100	-100
2.6 kHz	USB	200	2800
	LSB	-2800	-200
2.1 kHz	USB	200	2300
	LSB	-2300	-200
1000 Hz	USB	200	1200
	LSB	-1200	-200
500 Hz	USB	350	850
	LSB	-850	-350
250 Hz	USB	475	725
	LSB	-725	-475
100 Hz	USB	550	650
	LSB	-650	-550
50 Hz	USB	575	625
	LSB	-625	-575
25 Hz	USB	587	613
	LSB	-613	-587

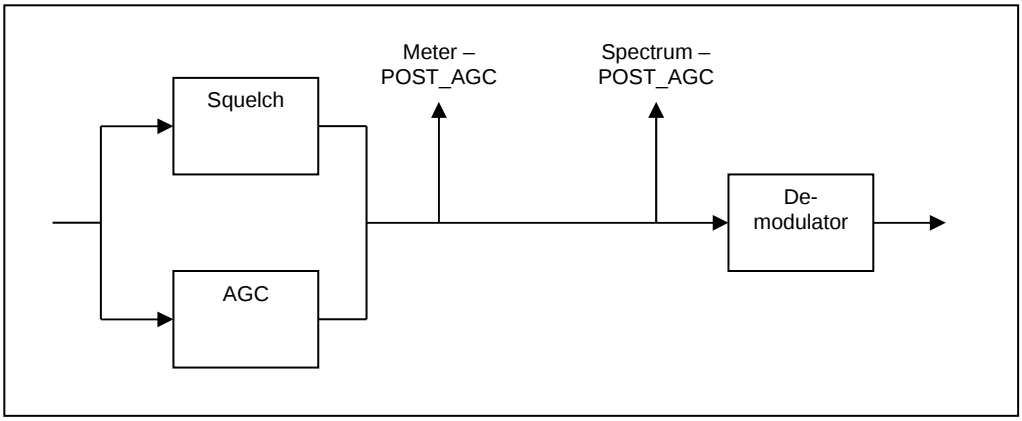
It is possible to use the narrower bandwidths on AM and FM, however their usefulness is debatable and have not been included in the tables.

At this stage the signal is sampled for both the meter and for the spectrum display. The meter is updated via `do_rx_meter()` and the information is available as meter types `SIGNAL_STRENGTH` and `AVG_SIGNAL_STRENGTH`. The spectrum data is of type post filter, a full discussion of the different spectrum types if available above.

The Compander

The next stage is the optional receive compander. This is enabled via the pipe command "`setCompandSt`" with the value of the compander set by the "`setCompand`" pipe command. There is no equivalent DLL function to enable the receive compander. The compander is the function `WSCompand()` in the file `wscompand.c`. The default is for the compander to be off.

The compander is only of use if the transmit compander is also enabled. It acts as a cross of a speech processor and a Dolby system. Its aim is to reduce background noise on SSB transmissions.



The signal is then tested to see whether it should be squelched or hav AGC applied to it. The function `should_do_rx_squelch()` in `sdr.c` is called which tests the squelch flag, if it is true then average input signal is tested against the squelch value, if it is less then the squelch

will be applied. If the squelch flag is not set, or the signal is too strong then the function returns false and AGC is applied instead. The receive squelch flag is set by the DLL function `SetSquelchSt()` and the pipe command "`setSquelchSt`". The value of the squelch threshold is set by the DLL command `SetSquelchVal()` and the pipe command "`setSquelch`", the value of the threshold is a floating point number in decibels, the default value is –150.0.

Receive Squelch

If the squelch is applied then the function `do_squelch()` in `sdr.c` is called. This either sets the signal to zeros (i.e. silence) or ramps it to silence. The AGC is performed in the function `DttSPAgc()` in `dttspagc.c`. The AGC system is one of the most configurable subsystem in the whole of DttSP, with a great many parameters that can be set. Thankfully a simple means to set the AGC in a number of preset values is available, this is done via the DLL function `SetRXAGC()` and the pipe command "`setRXAGC`". Its values may be one of:

AGC Mode	Pipe Number
`agcOFF`	0
`agcLONG`	1
`agcSLOW`	2
`agcMED`	3
`agcFAST`	4

At this stage the signal is tapped to enable another meter reading (via `do_rx_meter()`) providing the value for the meter type of `AGC_GAIN`. The signal is then sampled for a spectrum display of type post agc.

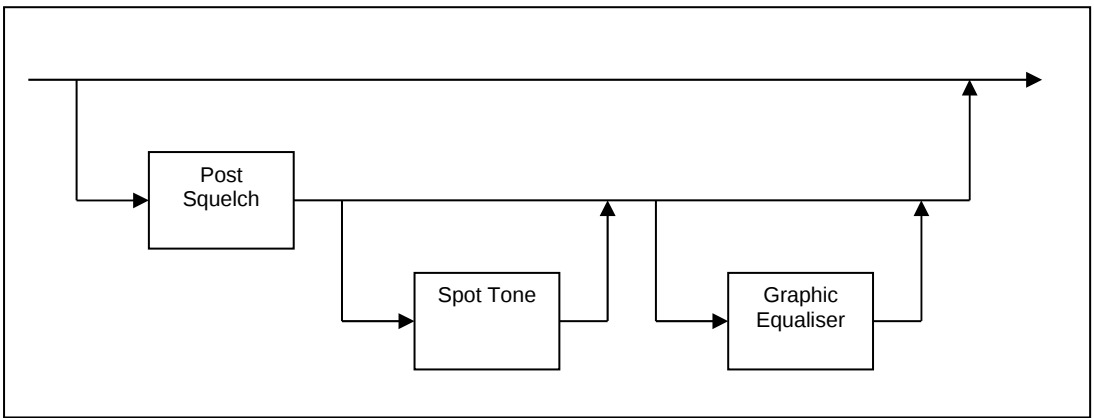
Demodulation

At this stage the function `do_rx_pre()` ends and returns to `do_rx()`. Here the correct demodulator is chosen depending on the mode setting. The receive mode is set by the DLL command `SetMode()` and the pipe command "`setMode`", the values allowed are:

Mode	Mode Number	Comments
`LSB`	0	
`USB`	1	
`DSB`	2	
`CWL`	3	CW using LSB
`CWU`	4	CW using USB
`FMN`	5	
`AM`	6	
`DIGU`	7	Digital modes using USB
`SPEC`	8	
`DIGL`	9	Digital modes using LSB
`SAM`	10	Synchronous AM detection
`DRM`	11	Digital radio mondiale

The description of the individual demodulators is in another section.

After the demodulator has processed the input signal, the function `do_rx_post()` in `sdr.c` is called to do the common final processing.



Initially the squelch is tested to see if it currently set, if not then `no_squelch()` in `sdr.c` which releases the output data gradually.

Spot Tone Generator

The next optional stage is the spot tone. This is only available when using the pipe commands and is enabled via the "`setSpotTone`" command, and the frequency of the tone is set by the "`setSpotToneVals`" command along with the rise and fall times and the gain. The tone is created in the `SpotTone()` function in spottone.c.

Graphic Equaliser

The final step in the receive processing is to optionally run the graphic equaliser. This is done in the `graphiceq()` function in `graphiceq.c`. It is enabled via the DLL function `SetGrphRXEQcmd()` and the pipe command "`setGrphRXEQcmd`", the values of the equaliser are set with the DLL function `SetGrphRXEQ()` which takes an array of four integers as an argument, and the pipe command "`setGrphRXEQ`" which likewise takes four floating point arguments. The graphic equaliser operates over a 12 kHz bandwidth, with the first argument being an overall gain setting (in dB), the second argument being the gain (in dB) to apply in the range –400 to +400 Hz. The third argument is the gain (in dB) for the range +400 to +1500 Hz and –400 to –1500 Hz, the final argument is the gain (in dB) for the range +1500 to +6000 Hz and –1500 to –6000 Hz.

The Transmit Chain

As you would imagine the transmit chain is very similar to the receive chain, only in reverse. The input signal, which only exists as a real signal, i.e. no quadrature component, is converted to a complex value with zero being set for the missing part. Because the input data is in the form Q I rather than I Q, the imaginary part of the input data is taken and placed into the real part.

DC Blocker

The first stage is the optional removal of any DC component of the signal. The DC blocker is an IIR highpass filter using the Butterworth polynomial with a knee frequency of 100 Hz. There are four filters to choose from with two, four, six and eight poles, by default the four pole filter is used. The DC blocker is set for a sample rate of 48 kHz and will therefore not be correct for other sample rates.

The DC blocking is performed by the function DCBlock() in file dcblock.cpp. The DC Blocker is enabled with the DLL function SetDCBlock() and the pipe command "setDCBlockSt", the default is off. The choice of the number of poles of filtering is set by the pipe command "setDCBlock", there is no equivalent DLL function.

DC Block Value	No of Poles	Level
0	2	Low
1	4	Medium
2	6	High

3	8	Super

The DC blocker should not be used for data modes, due to its use of an IIR filter. These filters, while good, do not preserve phase and would therefore cripple data modes such as PSK. Thankfully data modes should not include any DC component and the DC blocker is mainly reserved for use with microphone input. The DC blocker only operates on the real part of the input signal.

The FM Modulator and Demodulator

The AM Modulator and Demodulator

The SSB/CW Modulator and Demodulator

Other Modes

The two remaining modes are `SPEC` and `DRM`. In both cases they have no modulator or demodulator available, but only stub functions are used which do nothing. On receive the functions used are `do_rx_SPEC()` and `do_rx_DRM()` and on transmit, `do_tx_NIL()`, all are to be found in `sdr.c`.