PSKCore.DLL

Software Specification

and

Technical Guide

Ver. 1.20

Dec 24, 2000

by

Moe Wheatley, AE4JY
ae4jy@mindspring.com

Table of Contents

1. Introduction 3

1.1. KEY DLL FEATURES: 3

2. PSKCore.dll Software Interface Specification 5

2.1. INITIALIZATION/SHUTDOWN FUNCTIONS 5
2.1.1 fnStartSoundCard 5
2.1.2 fnStopSoundCard 6
2.2. RECEIVE FUNCTIONS 7
2.2.1 fnSetRXFrequency 7
2.2.2 fnSetRXPSKMode 7
2.2.3 fnGetRXFrequency 8
2.2.4 fnSetFFTMode 8
2.2.5 fnGetFFTData 9
2.2.6 fnGetClosestPeak 9
2.2.7 fnGetSyncData 10
2.2.8 fnGetVectorData 10
2.2.9 fnGetRawData 11
2.2.10 fnSetAFCLimit 11
2.2.11 fnSetSquelchThreshold 12
2.2.12 fnGetSignalLevel 12
2.2.13 fnRewindInput 13
2.3. TRANSMIT FUNCTIONS 14
2.3.1 fnStartTX 14
2.3.2 fnStopTX 14
2.3.3 fnAbortTX 14
2.3.4 fnSetTXFrequency 15
2.3.5 fnSetCWIDString 15
2.3.6 fnSendTXCharacter 15
2.3.7 fnSendTXString 16
2.3.8 fnGetTXCharsRemaining 16
2.3.9 fnClearTXBuffer 17
2.3.10 fnSetCWIDSpeed 17
2.3.11 fnSetComPort 17
2.4. MISCELLANEOUS FUNCTIONS 19
2.4.1 fnSetClockErrorAdjustment 19
2.4.2 fnGetDLLVersion 19
2.4.3 fnGetErrorString 19
2.5. USER WINDOW'S MESSAGE DEFINITIONS 20
2.5.1 MSG_DATARDY 20
2.5.2 MSG_PSKCHARRDY 20
2.5.3 MSG_STATUSCHANGE 20
2.5.4 MSG_IMDRDY 21
2.5.5 MSG_CLKERROR 21

3. Technical Operation Description 22

3.1. PSK31 SIGNAL GENERATION 22
3.1.1 Input Characters 22
3.1.2 Varicode Encoding 22
3.1.3 BPSK Serialization 24
3.1.4 QPSK Serialization 24
3.1.5 Differential Phase Shift encoding 25
3.1.6 Wave Shaping and Carrier Generation 26
3.1.7 Power Spectrum 30
3.2. PSK31 SIGNAL DETECTION 31
3.2.1 Block Diagram 31
3.2.2 Soundcard Input 32
3.2.3 Complex Mixer 32
3.2.4 Decimation by 16 33
3.2.5 Matched Data Bit filter 34
3.2.6 Frequency Error Filter 35
3.2.7 AGC 35
3.2.8 Frequency Error Detection/Correction 37
3.2.9 Symbol Synchronization 40
3.2.10 Squelch Function 41
3.2.11 IMD Measurement 45
3.2.12 Symbol Decoding 48

4. Further References and Acknowledgments 51

1. Introduction

PSK31 is an amateur radio communications mode introduced by Peter Martinez, G3PLX, that uses phase modulation and special character coding. It allows robust narrow bandwidth keyboard “Chat” type communications between two or more stations.

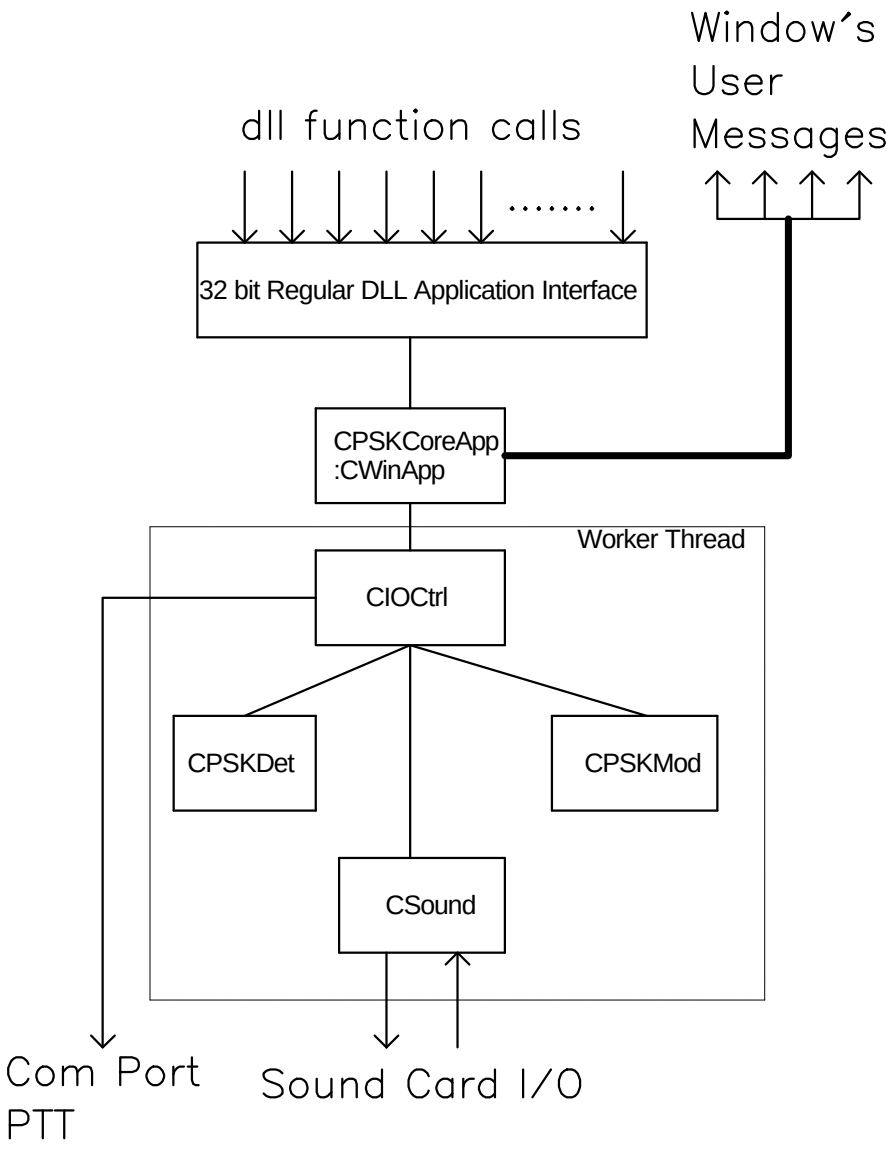
This document describes the programming interface to a DLL(Dynamic Link Library) that can be used for receiving and transmitting PSK31 signals using a Windows soundcard. The DLL is a 32 bit Regular DLL written using Visual C++ and MFC. The user sends and receives data and status via calls to various routines exposed by the DLL. Custom User Windows messages are sent from the DLL to the application for notification of events that require action by the application program.

The DLL is made available free of charge for amateur radio use only.

1.1. Key DLL Features:

- Interfaces to various applications environments including C, Visual C++, Visual Basic, and Delphi.
- Multiple soundcard support.
- Allows multiple independent receiver channels.
- Threshold adjustable noise activated squelch control.
- PSK31 Signal strength/quality value available for squelch and display.
- Soft decision Viterbi decoder for QPSK mode
- Symbol Vector tuning data for phase type display.
- Frequency Spectrum data available for Display 0 to 4000 Hz with 3.9Hz resolution.
- Log or 4^{th} root frequency spectrum data formats.
- Raw soundcard data mode.
- S/N qualified IMD measurement data.
- Soundcard clock error measuring and compensating capability.
- Includes Serial COM port RTS and/or DTR PTT control.
- CW ID capability with multiple speed selections.
- 20000 character transmit buffer.
- Full duplex Tx and RX can be invoked.
- A fast AFC mode can be used to track doppler shifting BPSK signals up to +/-20Hz/Sec.

Basic DLL Software Structure



2. PSKCore.dll Software Interface Specification

2.1. Initialization/Shutdown Functions

2.1.1 fnStartSoundCard

VC++ Prototype:
long __stdcall fnStartSoundCard(HWND hWnd, long cardnum, long numRXchannels);

VB Prototype:
Declare Function fnStartSoundCard Lib "PSKCore.dll" (ByVal hWnd As Long, ByVal cardnum As Long, ByVal numRXchannels As Long) As Long

Delphi Pascal Prototype:
function fnStartSoundCard(h_Wnd: hWnd; cardnum, numRXchannels: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- hWnd - A Handle to the Window that is to receive the DLL's messages.
- cardnum - number of soundcard to use(0 to 3) (use -1 to let the system locate the soundcard)
- numRXchannels - Number of Receive channels to use.(1, 2, or 3)

Return Value:

Returns a long value indicating error status.
0 = No error in starting sound card process.
10 = Memory Allocation error
11 = Tried to read and soundcard is not open
12 = Input buffers over-flowed
13 = Timed out waiting for input buffers
14 = Tried to write and soundcard is not open
15 = Output buffers under-flowed
16 = Timed out waiting for output buffers
17 = Card doesn't support 16bit, 8000Hz, Mono format
18 = Still something playing on soundcard
19 = Header not prepared
20 = Device is synchronous
21 = Bad Device ID, Soundcard Not Present
22 = Driver failed to enable
23 = Device already allocated
24 = Device handle is invalid
25 = No device driver present
26 = Function isn't supported
27 = Error value out of range
28 = Invalid flag passed
29 = Invalid parameter passed
30 = Handle being used
31 = Driver does not call DriverCallback
32 = Registry error
33 = Unknown Error

Comments:
Call this function to start up the soundcard process and begin processing PSK data.

2.1.2 fnStopSoundCard

VC++ Prototype:
void __stdcall fnStopSoundCard();

VB Prototype:
Declare Sub fnStopSoundCard Lib "PSKCore.dll" ()

Delphi Pascal Prototype:
procedure fnStopSoundCard; stdcall; external 'pskcore.dll';

Parameters:
none.

Return Value:
None.

Comments:
Call this function to stop the soundcard process and stop processing PSK data.
This must be called before exiting your program to ensure all the threads and resources are de-allocated.

2.2. Receive Functions

2.2.1 fnSetRXFrequency

VC++ Prototype:
void __stdcall fnSetRXFrequency (long freq, long range, long channel);

VB Prototype:
Declare Sub fnSetRXFrequency Lib "PSKCore.dll" (ByVal freq As Long, ByVal range As Long, ByVal channel As Long)

Delphi Pascal Prototype:
procedure fnSetRXFrequency(freq, range, channel: integer); stdcall; external 'pskcore.dll';

Parameters:

- freq = PSK31 Receive audio center frequency. (100 to 3500 Hz.) (default=1500)
- range = Search range in Hz. (0 to 50 Hz)
- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:
Called to set a new receive audio center frequency. The search range parameter is used to set limits on how far to search the fft data for a peak. When fnSetRXFrequency(..) is called, a frequency peak is searched for between (freq-range) and (freq+range). This peak frequency estimate is then used as the new center frequency for the receiver. The PSK31 receiver then begins fine tuning it's frequency within the AFC Limits which are set using the fnSetAFCLimit() function. A range of 0 disables the search mode.

2.2.2 fnSetRXPSKMode

VC++ Prototype:
void __stdcall fnSetRXPSKMode(long mode, long chan);

VB Prototype:
Declare Sub fnSetRXPSKModeLib "PSKCore.dll" (ByVal mode As Long, ByVal chan As Long)

Delphi Pascal Prototype:
procedure fnSetRXPSKMode(mode, chan: integer); stdcall; external 'pskcore.dll';

Parameters:

- mode = PSK31 Mode.
  0 = BPSK (default mode)
  1 = QPSK usb
  2 = QPSK lsb.
- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:
Called to set the PSK31 demodulation mode for RX channels.

2.2.3 fnGetRXFrequency

VC++ Prototype:
long __stdcall fnGetRXFrequency (long channel);

VB Prototype:
Declare Function fnGetRXFrequency Lib "PSKCore.dll" (ByVal channel As Long) As Long

Delphi Pascal Prototype:
function fnGetRXFrequency(channel: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- channel = Receive channel(0, 1, or 2)

Return Value:
The current Receive frequency.(100 to 3500)Hz.

Comments:
Call this to obtain the current receive center frequency. This is useful to keep up with the receive frequency as the AFC moves it around. The RX frequency for channel 0 is also sent back as a parameter with the MSG_DATARDY windows message.

2.2.4 fnSetFFTMode

VC++ Prototype:
void __stdcall fnSetFFTMode (long ave, long maxscale, long mode);

VB Prototype:
Declare Sub fnSetFFTMode Lib "PSKCore.dll" (ByVal ave As Long, ByVal maxscale As Long, ByVal mode As Long)

Delphi Pascal Prototype:
procedure fnSetFFTMode(ave, maxscale, mode: integer); stdcall; external 'pskcore.dll';

Parameters:

- ave = FFT "smoothing" value. (1 to 10) 1 = no smoothing, 10 = Max smoothing(default=1)

- maxscale = FFT amplitude scaling value. This is the maximum value the FFT will return if the input is a fullscale sinewave. (default = 100)

If maxscale is 100 and mode is log, then the the fft range is 0 to 100dB.
If maxscale is 2500, then the the fft range is 0 to 2500.

- mode = data output type.
  0 = Root mode. Data is the 4th root of the linear FFT output power
  1 = log mode. Data is 10log() of the FFT power. (default mode)
  10 to 90 = log mode with (10% to 90%) baseline clipping applied to the data.

Return Value:
None.

Comments:
Called to set FFT data mode, averaging mode, baseline clipping and amplitude scaling value. If the mode value is between 10 and 90, then the data is baseline clipped by that percentage. For example if the mode is 20 and maxscale is 100, then all data from 20 to 100 is shifted and scaled to range from 0 to 100. This can be used to shift the fft data so that the baseline noise is at the bottom of the viewing screen, giving a larger viewing dynamic range.

2.2.5 fnGetFFTData

VC++ Prototype:
long __stdcall fnGetFFTData (long* DataArray, long fstart, long fend);

VB Prototype:
Declare Function fnGetFFTData Lib "PSKCore.dll" (DataArray As Long, ByVal fstart As Long, ByVal fend As Long) As Long

Delphi Pascal Prototype:
function fnGetFFTData(DataArray: PDataArray; fstart, fend: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- DataArray = Pointer to an array of 1024 longs that will be filled with FFT data by the dll.
- fstart = Starting index to the array.(0 to 1022)
- fend = Ending index to the array. (1 to 1023)

Return Value:
A boolean value. FALSE(0) = No Input Signal Overload. TRUE(not 0) = Audio input signal is overloading the soundcard.

Comments:
This routine is called after receiving the Windows User message MSG_DATARDY from the dll. This message is sent to the application window when the next block of data is available from the sound card. The 1024 longs are the amplitudes of 1024 frequencies. The DataArray(0) is zero frequency(not very useful). The DataArray(499) corresponds to 1949Hz. The DataArray(1023) corresponds to 3996Hz. Each point steps by 3.90625Hz(8000/2048). Use the fstart and fend parameters to copy partial ranges into your array. This is useful for zooming where you don't need all the data points. If the routine returns TRUE, the input level to the soundcard is too high and must be reduced.

2.2.6 fnGetClosestPeak

VC++ Prototype:
long __stdcall fnGetClosestPeak (long fstart, long fend);

VB Prototype:
Declare Function fnGetClosestPeak Lib "PSKCore.dll" (ByVal fstart As Long, ByVal fend As Long) As Long

Delphi Pascal Prototype:
function fnGetClosestPeak (fstart, fend: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- fstart = Starting frequency to search from(100 to 3500)
- fend = Ending frequency limit to search to(100 to 3500)

Return Value:
Frequency peak if found otherwise it just returns the start frequency if no peak is found.

Comments:
This function can be called to find the nearest spectral peak frequency from the 'fstart' frequency to 'fend' frequency. It makes no distinction of signal type, just looks for a peak.

2.2.7 fnGetSyncData

VC++ Prototype:
void __stdcall fnGetSyncData (long* SyncArray, long channel);

VB Prototype:
Declare Sub fnGetSyncData Lib "PSKCore.dll" (SyncArray As Long, ByVal channel As Long)

Delphi Pascal Prototype:
procedure fnGetSyncData (SyncArray: PSyncArray; channel: integer); stdcall; external 'pskcore.dll';

Parameters:

- SyncArray = Pointer to an array of 16 longs that will be filled with signal sync data by the dll. Needs to be able to hold at least 16 longs.

- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:
Call this function to obtain a set of sync signals. Each value indicates the energy at each of the 16 sample points within a bit time(range 0 to 1000). The maximum energy bit time is used to determine the center of the bit and that is where the data is sampled. The usefulness of this is to be able to watch the bit center drift rate which indicates an off frequency soundcard either on the sending or receiving side.

2.2.8 fnGetVectorData

VC++ Prototype:
void __stdcall fnGetVectorData (long* VectorArray, long channel);

VB Prototype:
Declare Sub fnGetVectorData Lib "PSKCore.dll" (VectorArray As Long, ByVal channel As Long)

Delphi Pascal Prototype:
procedure fnGetVectorData(VectorArray: PVectorArray; channel: integer); stdcall; external 'pskcore.dll';

Parameters:

- VectorArray = Pointer to an array of 16 longs that will be filled with signal vector data by the dll. Needs to be able to hold at least 16 longs.

- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:
Call this function to obtain a set of signal vectors. Each 2 elements of the VectorArray[i] represent the x,y coordinates of a vector whose magnitude is 1000.
For example:

VectorArray[0] = 0,	VectorArray[1] = 1000	is a vector pointing up(steady carrier 0 deg.)
VectorArray[2] = 1000,	VectorArray[3] = 0	is a vector pointing right(+90deg.)
VectorArray[4] = 0,	VectorArray[5] = -1000	is a vector pointing down(idle 180deg)
VectorArray[6] = -1000,	VectorArray[7] = 0	is a vector pointing left(-90 deg.)
VectorArray[6] = -867,	VectorArray[7] = 500	is a -60 deg. vector

pseudo code example:
// xc,yc is the pixel coordinates of the center of the display circle.(y increases down the screeen)
// r is the radius in pixels of the display circle.

```
for(x=0; x<16; x+=2)
{
  MemDC.MoveTo( xc,  yc );
  MemDC.LineTo( xc + (r * VectorArray[x]/1000), yc - (r * VectorArray[x+1]/1000) );
}
```

2.2.9 fnGetRawData

VC++ Prototype:
long __stdcall fnGetRawData (long* DataArray, long dstart, long dend);

VB Prototype:
Declare Function fnGetRawData Lib "PSKCore.dll" (DataArray As Long, ByVal dstart As Long, ByVal dend As Long) As Long

Delphi Pascal Prototype:
function fnGetRawData(DataArray: PDataArray; dstart, dend: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- DataArray = Pointer to an array of 2048 longs that will be filled with raw input samples from the soundcard.
- dstart = Starting index to the array.(0 to 2046)
- dend = Ending index to the array. (1 to 2047)

Return Value:
A boolean value. FALSE(0) = No Input Signal Overload. TRUE(not 0) = Audio input signal is overloading the soundcard.

Comments:
This routine can be called in place of or in addition to fnGetFFTData after receiving the Windows User message MSG_DATARDY from the dll. This message is sent to the application window when the next block of data is available from the sound card.
The 2048 longs contain the raw signal amplitudes(-32768 to 32767) sampled at 8000Hz from the soundcard.

2.2.10 fnSetAFCLimit

VC++ Prototype:
void __stdcall fnSetAFCLimit (long limit, long channel);

VB Prototype:
Declare Sub fnSetAFCLimit Lib "PSKCore.dll" (ByVal limit As Long, ByVal channel As Long)

Delphi Pascal Prototype:
procedure fnSetAFCLimit (limit, channel: integer); stdcall; external 'pskcore.dll';

Parameters:

- limit = AFC limit in Hz. (0 to 1000, or 3000 Hz) PSK center frequency will be bounded +/-limit Hz(default=50). If set to 3000Hz, then a special fast AFC is used for tracking doppler shifting signals.
- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:

Called to set the AFC frequency limit of operation. A limit of = 0 turns off any AFC action.

2.2.11 fnSetSquelchThreshold

VC++ Prototype:
void __stdcall fnSetSquelchThreshold (long thresh, long mode, long channel);

VB Prototype:
Declare Sub fnSetSquelchThreshold Lib "PSKCore.dll" (ByVal thresh As Long, ByVal mode As Long, ByVal channel As Long)

Delphi Pascal Prototype:
procedure fnSetSquelchThreshold(thresh, mode, channel: integer); stdcall; external 'pskcore.dll';

Parameters:

- thresh = A value between 0 and 99 to set the squelch threshold. (defualt=50)
- mode = Squelch response speed
  0 = Fast(uses fixed value for fast filtering)
  1 = Slow(uses fixed value for slow filtering) (default speed)
  10-200 = User selectable response speed.(10 is fastest, 200 is slowest response)
- channel = Receive channel(0, 1, or 2)

Return Value:
None.

Comments:
Called to set the squelch threshold. If a signal value exceeds this threshold, PSK31 characters will be decoded and made available to the application. A value of zero will decode characters regardless of the signal strength (Open Squelch). The mode parameter can be used to select a fast response useful under good signal conditions or slow which operates better under noisy conditions or any custom user selectable speed.

2.2.12 fnGetSignalLevel

VC++ Prototype:
long __stdcall fnGetSignalLevel (long channel);

VB Prototype:
Declare Function fnGetSignalLevel Lib "PSKCore.dll" (ByVal channel As Long) As Long

Delphi Pascal Prototype:
function fnGetSignalLevel(channel: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- channel = Receive channel(0, 1, or 2)

Return Value:
A value from 0 to 99 that is an indicator of signal quality.

Comments:
Called to get the signal quality. This value is compared with the squelch threshold to determine whether to decode characters or not. Useful for providing a signal strength/quality indicator. The signal strength for receive channel 0 is also sent back as a parameter with the MSG_DATARDY windows message. This value is not the amplitude of the signal but a measure of how much phase noise is on the incoming signal.

2.2.13 fnRewindInput

VC++ Prototype:
void __stdcall fnRewindInput (long blocks);

VB Prototype:
Declare Sub fnRewindInput Lib "PSKCore.dll" (ByVal blocks As Long) As Long

Delphi Pascal Prototype:
procedure fnRewindInput (channel: blocks); stdcall; external 'pskcore.dll';

Parameters:

- blocks = Number of data blocks to back up.(1 to 99) Each block is 2048 samples or .256 Seconds.

Return Value:
None

Comments:
Called to rewind the dll's input audio data. This can be used to back up in time and re-decode signals that have already been received. The dll keeps 99 blocks of past data that can be replayed. Each block is equivalent to one FFT buffers worth and is .256 Seconds per block giving a maximum rewind time of 25.344 Seconds. When this function is called, the replay occurs about 50 times as fast(depending on CPU speed) as normal so all the data ready and decoded characters will be sent at a much faster rate until the decoder catches up with the real time data again.

2.3. Transmit Functions

2.3.1 fnStartTX

VC++ Prototype:
void __stdcall fnStartTX (long mode);

VB Prototype:
Declare Sub fnStartTX Lib "PSKCore.dll" (ByVal mode As Long)

Delphi Pascal Prototype:
procedure fnStartTX(mode: integer); stdcall; external 'pskcore.dll';

Parameters:

- mode - Type of transmission.
  0 = BPSK
  1 = QPSK usb
  2 = QPSK lsb
  3 = Tune(steady carrier)
  4 = Tune(steady carrier) follow with CWID.
  if above parameter has bit 7 set, then the receiver is left on and full duplex operation is available.

Return Value:
None.

Comments:
Call this function to change from RX to TX with the specified transmit mode. The soundcard stops receiving audio and begins outputting audio. If the COM port PTT is enabled, The RTS and/or DTR lines will become active.

2.3.2 fnStopTX

VC++ Prototype:
void __stdcall fnStopTX();

VB Prototype:
Declare Sub fnStopTX Lib "PSKCore.dll" ()

Delphi Pascal Prototype:
procedure fnStopTX; stdcall; external 'pskcore.dll';

Parameters:
none.

Return Value:
None.

Comments:
Call this function to change from TX mode back to RX mode. Any characters left to transmit and any CW ID is completed before switching back to RX.

2.3.3 fnAbortTX

VC++ Prototype:
void __stdcall fnAbortTX ();

VB Prototype:
Declare Sub fnAbortTX Lib "PSKCore.dll" ()

Delphi Pascal Prototype:

procedure fnAbortTX; stdcall; external 'pskcore.dll';

Parameters:
none.

Return Value:
None.

Comments:
Call this function to change from TX mode back to RX mode. Any pending characters or CW ID is aborted and the dll goes back to RX mode. Calling this function when in receive mode will clear all transmit characters from the TX buffer.

2.3.4 fnSetTXFrequency

VC++ Prototype:
void __stdcall fnSetTXFrequency (long freq);

VB Prototype:
Declare Sub fnSetTXFrequency Lib "PSKCore.dll" (ByVal freq As Long)

Delphi Pascal Prototype:
procedure fnSetTXFrequency(freq: integer); stdcall; external 'pskcore.dll';

Parameters:
- freq = PSK31 Transmit audio center frequency. (100 to 3500 Hz.) (default=1500)

Return Value:
None.

Comments:
Called to set a new Transmit audio center frequency.

2.3.5 fnSetCWIDString

VC++ Prototype:
void __stdcall fnSetCWIDString (char* IDStrg);

VB Prototype:
Declare Sub fnSetCWIDString Lib "PSKCore.dll" (ByVal IDStrg As String)

Delphi Pascal Prototype:
procedure fnSetCWIDString(IDStrg: PChar); stdcall; external 'pskcore.dll';

Parameters:
- IDStrg = Null terminated string containing the string to send as a CW ID at the end of a PSK31 transmission. For prosign characters, '*' is SK, '+' is AR, and '=' is BT. (default="Call Not Set")

Return Value:
None.

Comments:
Called to specify a character string to send as a CW ID at the end of a transmission.

2.3.6 fnSendTXCharacter

VC++ Prototype:
long __stdcall fnSendTXCharacter (long txchar, long cntrl);

VB Prototype:
Declare Function fnSendTXCharacter Lib "PSKCore.dll" (ByVal txchar As Long, ByVal cntrl As Long) As Long

Delphi Pascal Prototype:
function fnSendTXCharacter(txchar, control: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- txchar = ASCII Character to be Transmitted(0 to 255) or control codes.
  Transmitter control codes:
  1 = AutoStopTX control code.
  2 = Add CWID at end of TX control code.

- cntrl = Control Code flag
  FALSE = 0 = txchar is ASCII character to send.
  TRUE = Not 0 = txchar is a transmitter control code.

Return Value:
The number of ASCII characters left in the TX buffer that have not yet been sent.

Comments:
Called to send an ASCII character. The characters are buffered up so this routine can be called without waiting for the character to be sent. It can be called prior to beginning transmission for type ahead buffering. If the cntrl parameter is TRUE(not zero), then the txchar parameter is interpreted as a control code for adding a CWID, invoking the TX autostop feature when the tx buffer goes empty, etc. If txchar is a 'Backspace(8)' character and there is at least one other character in the Transmit buffer that has not yet been transmitted, then the last character in the buffer will be removed and not sent. If there is no other characters in the buffer, then the Backspace(8) code will be transmitted so the receiving side can handle it.

2.3.7 fnSendTXString

VC++ Prototype:
long __stdcall fnSendTXString(char* lpszTXStrg);

VB Prototype:
Declare Function fnSendTXStringLib "PSKCore.dll" (ByVal TXStrg As String) As Long

Delphi Pascal Prototype:
function fnSendTXString(lpszTXStrg: PChar): integer; stdcall; external 'pskcore.dll';

Parameters:

- TXStrg = Null terminated ASCII string to send to the Transmitter.

Return Value:
The number of ASCII characters left in the TX buffer that have not yet been sent.

Comments:
Called to send a character string.

2.3.8 fnGetTXCharsRemaining

VC++ Prototype:
long __stdcall fnGetTXCharsRemaining ();

VB Prototype:
Declare Function fnGetTXCharsRemaining Lib "PSKCore.dll" () As Long

Delphi Pascal Prototype:
function fnGetTXCharsRemaining: integer; stdcall; external 'pskcore.dll';

Parameters:
None.

Return Value:
The number of ASCII characters left in the TX buffer that have not yet been sent.

Comments:
Call this to obtain the number of TX characters not yet sent that are still remaining in the TX buffer.

2.3.9 fnClearTXBuffer

VC++ Prototype:
void __stdcall fnClearTXBuffer();

VB Prototype:
Declare Sub fnClearTXBuffer Lib "PSKCore.dll" ()

Delphi Pascal Prototype:
procedure fnClearTXBuffer; stdcall; external 'pskcore.dll';

Parameters:
None.

Return Value:
None.

Comments:
Called to clear the TX character buffer.

2.3.10 fnSetCWIDSpeed

VC++ Prototype:
void __stdcall fnSetCWIDSpeed (long speed);

VB Prototype:
Declare Sub fnSetCWIDSpeed Lib "PSKCore.dll" (ByVal speed As Long)

Delphi Pascal Prototype:
procedure fnSetCWIDSpeed(speed: integer); stdcall; external 'pskcore.dll';

Parameters:

- speed = CW ID Speed selector code:
  1 = 37.5 wpm.
  2 = 18.75 wpm.
  3 = 12.5 wpm.
  4 = 9.375 wpm.

Return Value:
None.

Comments:
Called to select a CWID speed. (wpm = 37.5/speed) for speed > 0

2.3.11 fnSetComPort

VC++ Prototype:
long __stdcall fnSetComPort (long portnum, long mode);

VB Prototype:
Declare Function fnSetComPort Lib "PSKCore.dll" (ByVal portnum As Long, ByVal mode as Long) As Long

Delphi Pascal Prototype:
function fnSetComPort(portnum, mode: integer): integer; stdcall; external 'pskcore.dll';

Parameters:

- portnum = Serial Com port number(1 to 8).

- mode = PTT mode:
  0 = No serial port PTT. (default mode)
  1 = Use RTS only.
  2 = Use DTR only.
  3 = Use RTS and DTR.

Return Value:
A boolean value. FALSE(0) = Specified Com port not available. TRUE(not 0) = Com port opened Okay.

Comments:
Called to open a serial Com port for use as PTT control. Can be called just to check to see which ports are available by looking at the return value. Also calling with any port number with a mode of '0' will deactivate the serial port PTT function.

2.4. Miscellaneous Functions

2.4.1 fnSetClockErrorAdjustment

VC++ Prototype:
void __stdcall fnSetClockErrorAdjustment (long ppm);

VB Prototype:
Declare Sub fnSetClockErrorAdjustment Lib "PSKCore.dll" (ByVal ppm As Long)

Delphi Pascal Prototype:
procedure fnSetClockErrorAdjustment(ppm: integer); stdcall; external 'pskcore.dll';

Parameters:

- ppm = Soundcard clock error adjustment value in parts per million. Range is +/- 10000 .(default=0)
  + ppm makes soundcard clock faster.
  - ppm makes soundcard clock slower.

Return Value:
None.
Comments:
Called to compensate for soundcards that are off frequency.

2.4.2 fnGetDLLVersion

VC++ Prototype:
long __stdcall fnGetDLLVersion ();

VB Prototype:
Declare Function fnGetDLLVersion Lib "PSKCore.dll" () As Long

Delphi Pascal Prototype:
function fnGetDLLVersion: integer; stdcall; external 'pskcore.dll';

Parameters:
None.
Return Value:
A long containing the DLL software revision number.(100 = 1.00, 123 = 1.23, etc)
Comments:
Called to get the DLL's software revision number.

2.4.3 fnGetErrorString

VC++ Prototype:
void __stdcall fnGetErrorString(char* ErrorStr);

VB Prototype:
Declare Sub fnGetErrorString Lib "PSKCore.dll" (ByVal ErrorStr As String)

Delphi Pascal Prototype:
procedure fnGetErrorString(ErrorStr: PChar); stdcall; external 'pskcore.dll';

Parameters:

- ErrorStr = Pointer to Null terminated ASCII string. The DLL will fill in this string with the latest error message. (ErrorString needs to point to at least 50 characters worth of buffer to hold the message string.)

Return Value:
None
Comments:
Called to retrieve any text details about the last error.

2.5. USER WINDOW'S MESSAGE DEFINITIONS

2.5.1 MSG_DATARDY

Numeric Value is WM_USER+1000 or 0x400+0x3E8 or 0x7E8 or 2024.

This message is sent from the DLL to the Window whose handle is passed when the fnStartSoundCard is called. It is sent whenever there is new data available from the FFT or Raw data from the soundcard. The message is sent every 0.256 Seconds while receiving and transmitting. It can also be used for general timing by the application for display updates, etc.

The following parameters are sent along with this message:

- wParam = current RX center frequency in Hz for RX channel 0.
- lParam = current signal strength(0 to 99)for RX channel 0.

Call fnGetFFTData and/or fnGetRawData in response to this message to obtain the data.

2.5.2 MSG_PSKCHARRDY

Numeric Value is WM_USER+1001 or 0x400+0x3E9 or 0x7E9 or 2025.

This message is sent from the DLL to the Window whose handle is passed when the fnStartSoundCard is called. It is sent whenever there is an ASCII character available from the receiver or if in the Transmit mode, when a character has been sent out the soundcard.

The following parameters are sent along with this message:

- wParam = The ASCII character(0 to 255)
- lParam = -1 if is a transmitted character, or the Receive channel number(0, 1, or 2) that is sending the message.

2.5.3 MSG_STATUSCHANGE

Numeric Value is WM_USER+1002 or 0x400+0x3EA or 0x7EA or 2026.

This message is sent from the DLL to the Window whose handle is passed when the fnStartSoundCard is called. It is sent whenever a status change occurs in the dll. The current status is available in the message parameters.

The following parameters are sent along with this message:

- wParam = status code.

Status Codes:

0 = In RX mode
1 = In TX mode
2 = CPU too slow or busy(soundcard was reset because CPU couldn't keep up)
3 = Is finishing transmitting remaining characters in buffer and any CW ID.
10 = Memory Allocation error
11 = Tried to read and soundcard is not open
12 = Input buffers over-flowed
13 = Timed out waiting for input buffers
14 = Tried to write and soundcard is not open
15 = Output buffers under-flowed
16 = Timed out waiting for output buffers
17 = Card doesn't support 16bit, 8000Hz, Mono format
18 = Still something playing on soundcard
19 = Header not prepared

20 = Device is synchronous
21 = Bad Device ID, Soundcard Not Present
22 = Driver failed to enable
23 = Device already allocated
24 = Device handle is invalid
25 = No device driver present
26 = Function isn't supported
27 = Error value out of range
28 = Invalid flag passed
29 = Invalid parameter passed
30 = Handle being used
31 = Driver does not call DriverCallback
32 = Registry error
33 = Unknown Error

- lParam = Not used.

2.5.4 MSG_IMDRDY

Numeric Value is WM_USER+1003 or 0x400+0x3EB or 0x7EB or 2027.

The following parameters are sent along with this message:

- wParam = The last calculated Received IMD.(0 to -100)dB.

- lParam = channel number ORed with 0x80 if the noise floor is higher than the IMD.
  channel = Receive channel(0, 1, or 2) if the IMD is a valid reading.
  channel = Receive channel(128, 129, or 130) if the IMD is the noise floor reading.

Comments:
This message is sent from the DLL to the Window whose handle is passed when the fnStartSoundCard is called. IMD is only calculated during periods of "idle" signals. If the channel number has bit 7 set, then the IMD reading is actually the noise floor. One can either choose to not display an IMD reading if this bit is set, or display a message that the actual IMD is less than this IMD reading. An example would be that the actual signal has a good -40dB IMD. Because the noise level is -20dB the IMD reading will be -20dB giving an incorrect reading. See the technical description about IMD for further explaination.

2.5.5 MSG_CLKERROR

Numeric Value is WM_USER+1004 or 0x400+0x3EC or 0x7EC or 2028.

The following parameters are sent along with this message:
wParam = Clock error in ppm.
+ error if soundcard clock is faster than the incoming signal clock.
- error if soundcard clock is slower than the incoming signal clock.

- lParam = RX channel number.

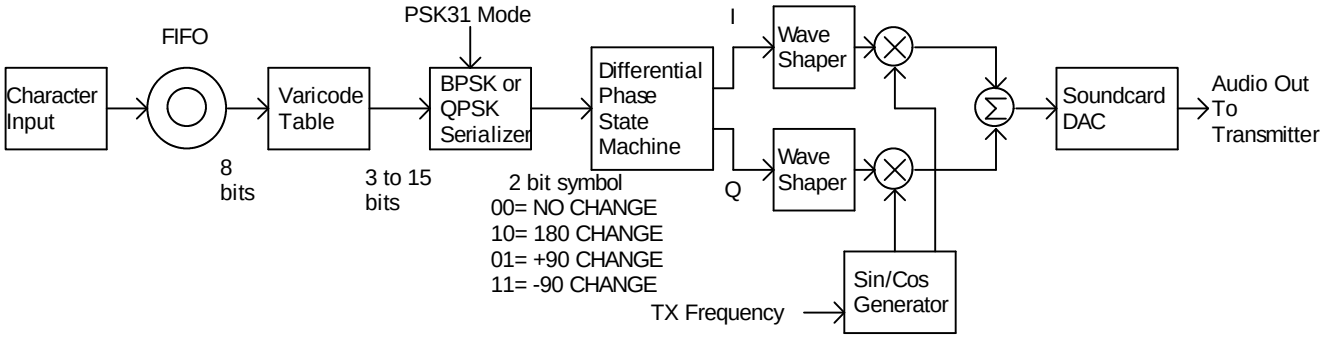
Comments:
This message is sent from the DLL to the Window whose handle is passed when the fnStartSoundCard is called. This message is only sent after 10 seconds of continuous open squelch reception. If there is no valid signal and the squelch is forced open, then the error value will not be meaningful. (Error can be due to the transmitted signal or one's own soundcard)

3. Technical Operation Description

3.1. PSK31 Signal Generation

Creating a PSK31 signal is done in stages starting with character input to final waveform output to the soundcard.

Overall Block Diagram of the PSK31 Transmit Section.



3.1.1 Input Characters

PSK31 sends and receives 8 bit characters. 0 through 127 are the standard ASCII characters and 128 to 255 are extended characters. The PSKCore dll has provisions for outputting a steady carrier tone for tuning as well as a CW ID mode that appends a Morse code string to the end of a transmission.

3.1.2 Varicode Encoding

The first step in PSK31 encoding is to map the 8 bit fixed length input characters into variable length characters. By mapping most used characters into shorter codes and least used characters into longer codes, the overall data transfer speed can be increased. This is similar to Morse code where common letters are shorter sequences. The letter 'e' occurs more often in text than a 'z' so it has a varicode of '11' while a 'z' has a code of '111010101'. Notice that lowercase letters have shorter codes than upper case letters. This is why one should not use all uppercase when using PSK31 since the varicode was optimized for lowercase letters.

Since the character data is sent serially, some means of separating characters is also needed. This is accomplished in PSK31 by specifying that two or more consecutive zero bits separate each character. This also places the requirement that each character code cannot contain more than one consecutive zero. It also means each code must start and end with a one. With these requirements the Varicode code table was specified. The varicode words from the table are sent msb first. If a new character is not ready in time to be sent, Zeros are padded into the data stream.

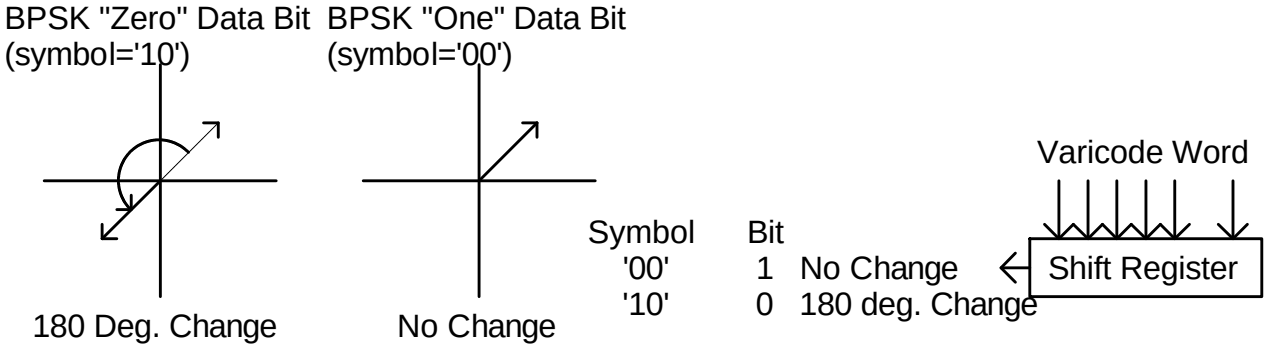
Example bit stream of varicoded character sequence "abc":

…00<u>1011</u>00<u>1011111</u>00<u>10111</u>00…..
a b c

Input Code	Varicode Output	Input Code	Varicode Output	Input Code	Varicode Output	Input Code	Varicode Output
NULL	1010101011	'@'	1010111101	128	1110111101	192	11011101111
SOH	1011011011	'A'	1111101	129	1110111111	193	11011110101
STX	1011101101	'B'	11101011	130	1111010101	194	11011110111
ETX	1101110111	'C'	10101101	131	1111010111	195	11011111011
EOT	1011101011	'D'	10110101	132	1111011011	196	11011111101
ENQ	1101011111	'E'	1110111	133	1111011101	197	11011111111
ACK	1011101111	'F'	11011011	134	1111011111	198	11101010101
BEL	1011111101	'G'	11111101	135	1111101011	199	11101010111
BS	1011111111	'H'	101010101	136	1111101101	200	11101011011
HT	11101111	'I'	1111111	137	1111101111	201	11101011101
LF	11101	'J'	111111101	138	1111110101	202	11101011111
VT	1101101111	'K'	101111101	139	1111110111	203	11101101011
FF	1011011101	'L'	11010111	140	1111111011	204	11101101101
CR	11111	'M'	10111011	141	1111111101	205	11101101111
SO	1101110101	'N'	11011101	142	1111111111	206	11101110101
SI	1110101011	'O'	10101011	143	10101010101	207	11101110111
DLE	1011110111	'P'	11010101	144	10101010111	208	11101111011
DC1	1011110101	'Q'	111011101	145	10101011011	209	11101111101
DC2	1110101101	'R'	10101111	146	10101011101	210	11101111111
DC3	1110101111	'S'	1101111	147	10101011111	211	11110101011
DC4	1101011011	'T'	1101101	148	10101101011	212	11110101101
NAK	1101101011	'U'	101010111	149	10101101101	213	11110101111
SYN	1101101101	'V'	110110101	150	10101101111	214	11110110101
ETB	1101010111	'W'	101011101	151	10101110101	215	11110110111
CAN	1101111011	'X'	101110101	152	10101110111	216	11110111011
EM	1101111101	'Y'	101111011	153	10101111011	217	11110111101
SUB	1110110111	'Z'	1010101101	154	10101111101	218	11110111111
ESC	1101010101	'['	111110111	155	10101111111	219	11111010101
FS	1101011101	'\'	111101111	156	10110101011	220	11111010111
GS	1110111011	']'	111111011	157	10110101101	221	11111011011
RS	1011111011	'^'	1010111111	158	10110101111	222	11111011101
US	1101111111	'_'	101101101	159	10110110101	223	11111011111
SPACE	1	'`'	1011011111	160	10110110111	224	11111101011
'!'	111111111	'a'	1011	161	10110111011	225	11111101101
'"'	101011111	'b'	1011111	162	10110111101	226	11111101111
'#'	111110101	'c'	101111	163	10110111111	227	11111110101
'$'	111011011	'd'	101101	164	10111010101	228	11111110111
'%'	1011010101	'e'	11	165	10111010111	229	11111111011
'&'	1010111011	'f'	111101	166	10111011011	230	11111111101
'''	101111111	'g'	1011011	167	10111011101	231	11111111111
'('	11111011	'h'	101011	168	10111011111	232	101010101011
')'	11110111	'i'	1101	169	10111101011	233	101010101101
'*'	101101111	'j'	111101011	170	10111101101	234	101010101111
'+'	111011111	'k'	10111111	171	10111101111	235	101010110101
','	1110101	'l'	11011	172	10111110101	236	101010110111
'-'	110101	'm'	111011	173	10111110111	237	101010111011
'.'	1010111	'n'	1111	174	10111111011	238	101010111101
'/'	110101111	'o'	111	175	10111111101	239	101010111111
'0'	10110111	'p'	111111	176	10111111111	240	101011010101
'1'	10111101	'q'	110111111	177	11010101011	241	101011010111
'2'	11101101	'r'	10101	178	11010101101	242	101011011011
'3'	11111111	's'	10111	179	11010101111	243	101011011101
'4'	101110111	't'	101	180	11010110101	244	101011011111
'5'	101011011	'u'	110111	181	11010110111	245	101011101011
'6'	101101011	'v'	1111011	182	11010111011	246	101011101101
'7'	110101101	'w'	1101011	183	11010111101	247	101011101111
'8'	110101011	'x'	11011111	184	11010111111	248	101011110101
'9'	110110111	'y'	1011101	185	11011010101	249	101011110111
':'	11110101	'z'	111010101	186	11011010111	250	101011111011
';'	110111101	'{'	1010110111	187	11011011011	251	101011111101
'<'	111101101	'|'	110111011	188	11011011101	252	101011111111
'='	1010101	'}'	1010110101	189	11011011111	253	101101010101
'>'	111010111	'~'	1011010111	190	11011101011	254	101101010111
'?'	1010101111	DEL	1110110101	191	11011101101	255	101101011011

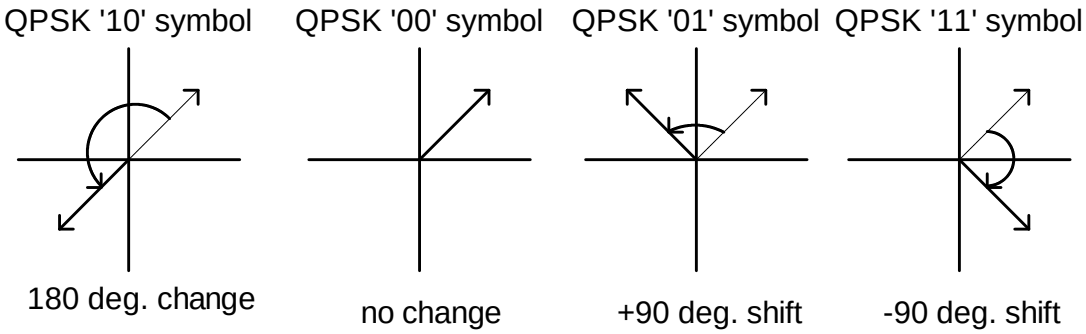
3.1.3 BPSK Serialization

PSK31 is actually Differential Phase Shift Keying because the information is sent as changes in signal phase rather than an absolute phase state. This makes signal reception much easier since the initial signal phase does not have to be known. For the Binary Phase Shift Keying mode, the signal either changes phase by 180 degrees for each ZERO bit or remains the same to represent a ONE bit. The symbol rate for PSK31 is 31.25 symbols per second or a period of .032 Seconds. The Varicode word is serialized and converted into a 2 bit symbol before being sent to the differential phase state machine which will determine the next signal phase based on the present phase and the new symbol.



3.1.4 QPSK Serialization

Quad Phase Shift Keying allows 4 unique phase states for each symbol effectively doubling the amount of information that can be sent over BPSK. Rather than send data twice as fast, PSK31 uses the extra information to allow for error correction.



3.1.4.1 ECC Encoding Method

The error correcting coding method used in PSK31 uses convolution codes to essentially "spread out" the redundant information over time. If one were to simply send each bit twice it is easily seen that if an error occurs in one of the bits, there is no way to tell which bit is the correct one so the redundancy is useless. If however the redundancy is spread out over several bits, there are some powerful mathematical methods to determine where the error occurred and correct it. Many books have been written to describe these methods so they will not be dealt with in any depth here.

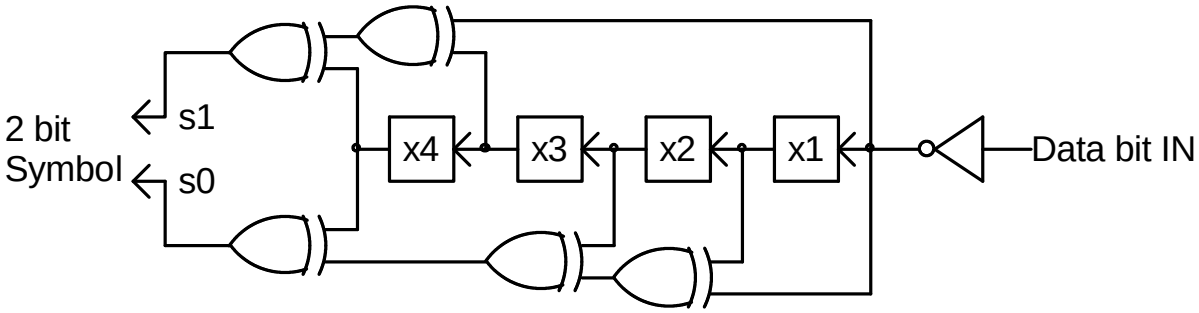
PSK31 spreads the data over 5 bits using rate ½ , constraint length 5, convolutional coding. The rate ½ refers to the fact that half of the data is being used for redundancy. The constraint length specifies the number of bits used to spread the redundancy.

Logically, a shift register is used to shift in each data bit. By exclusive OR'ing certain bits together, the desired symbol encoding is performed. The bit patterns(polynomials) which are used for exclusive OR'ing determines how well the system will be able to correct errors. The two polynomials used in PSK31 are:

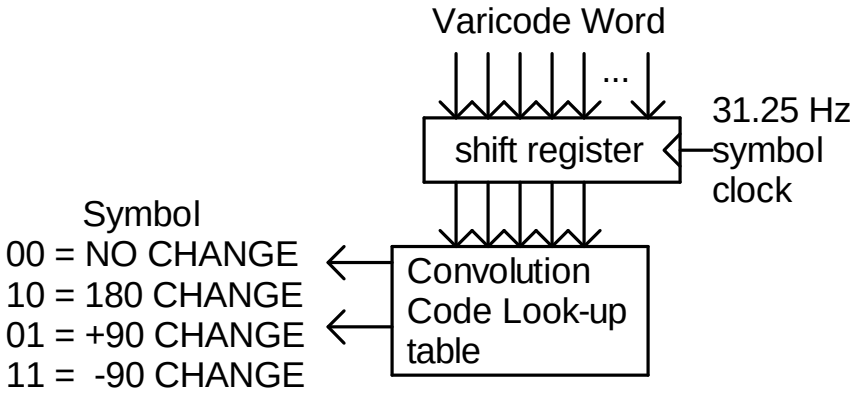
$$G1(x) = x^4 + x^3 + x^0$$

$$G0(x) = x^4 + x^2 + x^1 + x^0$$

The following diagram shows how the polynomials are used to generate a two bit symbol for every input bit.



Note that the data bits from the varicode word are inverted before entering the shift register. This is so that the idle stream of all zeros will produce symbols of '10' which are 180 degree phase shifts. This is useful for maintaining symbol sync on the receiver side and being compatible with BPSK.



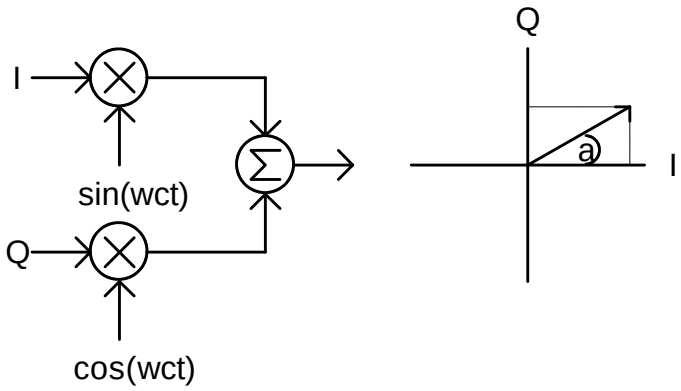
The QPSK encoder is actually implemented using a look-up table rather than using exclusive OR gates.

3.1.5 Differential Phase Shift encoding

The next step is to take the two bit symbol and convert it into the actual signal phase state. Depending on the previous signal phase, there are 4 signal phase possibilities for each new symbol. A simple state machine takes the present phase state information and the new symbol to come up with the next signal phase state. In PSKCore this is done using state tables.

3.1.6 Wave Shaping and Carrier Generation

The common way to create angle modulated signals is to combine two sinusoidal waveforms whose frequency is the desired carrier frequency and that are 90 degrees out of phase from each other. By adding these two signals in different proportions, a signal of any desired phase can be created. The two signals are referred to as I(in phase) and Q(quadrature phase).



The following MathCad simulation shows how a BPSK signal could be created using the I/Q method.

carrier amplitude $A := \frac{1}{\sqrt{2}}$

Symbol frequency $Fs := 31.25$

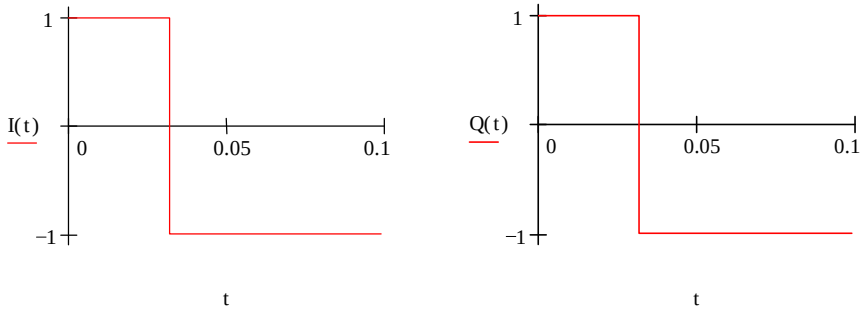
carrier frequency $Fc := 150$

Carrier equations

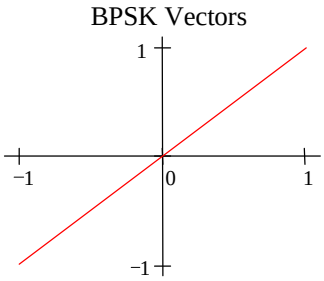
$Ic(t) := A \cdot \sin(2 \cdot \pi \cdot t \cdot Fc)$ $Qc(t) := A \cdot \cos(2 \cdot \pi \cdot t \cdot Fc)$

Modulation equations

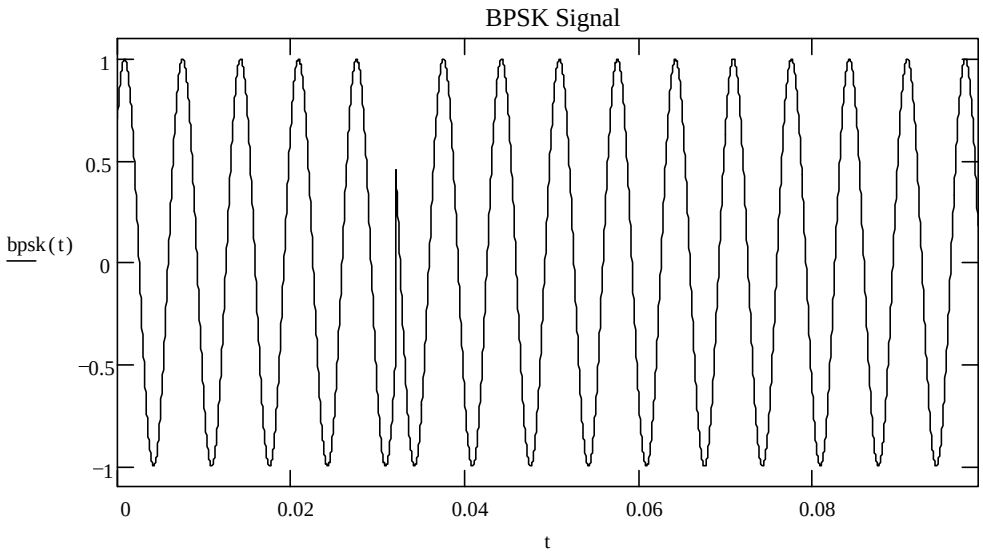
$I(t) := if((t<.032), 1, -1)$ $Q(t) := if((t<.032), 1, -1)$



This is a 180 degree phase shift followed by No phase shift.



$$bpsk(t) := I(t)\cdot Ic(t) + Q(t)\cdot Qc(t)$$



Note the abrupt phase change at time t = .032 seconds. This is not desirable since it makes the PSK signal very wide. One way to limit the bandwidth would be to filter the output signal. PSK31 uses a different method by using waveshaping on the I and Q input signals so that instead of abruptly going from a 1 to a –1, the signal makes a cosine shaped transition between –1 and 1.

Here is the same MathCad simulation except that the I and Q modulation signals are no longer rectangular, but are cosine shaped.

$Fc := 400$ carrier frequency $\qquad A := \frac{1}{\sqrt{2}}$ carrier amplitude

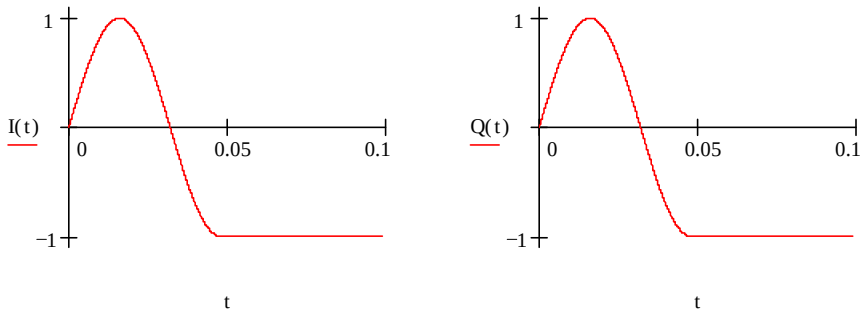
$Fs := 31.25$ Symbol frequency $\qquad t := 0, .000032 .. 0.099$ Plot range

carrier equations

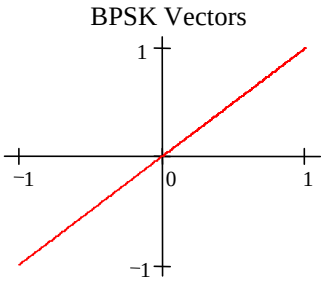
$Ic(t) := A\cdot\sin(2\cdot\pi\cdot t\cdot Fc) \qquad Qc(t) := A\cdot\cos(2\cdot\pi\cdot t\cdot Fc)$

modulation equations

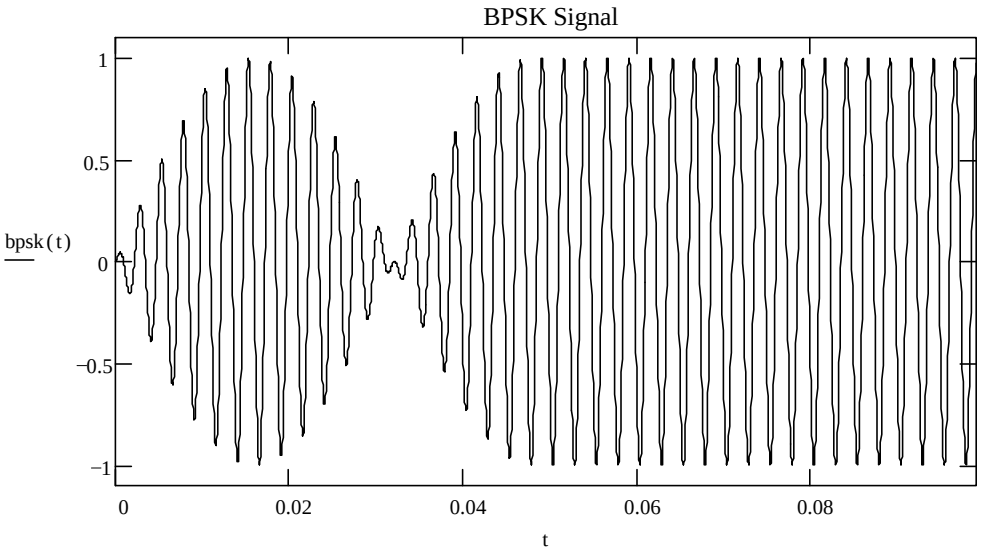
$I(t) := if((t<.048), \sin(\pi\cdot Fs\cdot t), -1) \qquad Q(t) := if((t<.048), \sin(\pi\cdot Fs\cdot t), -1)$



This is a 180 degree phase shift followed by No phase shift.



$$bpsk(t) := I(t)\cdot Ic(t) + Q(t)\cdot Qc(t)$$

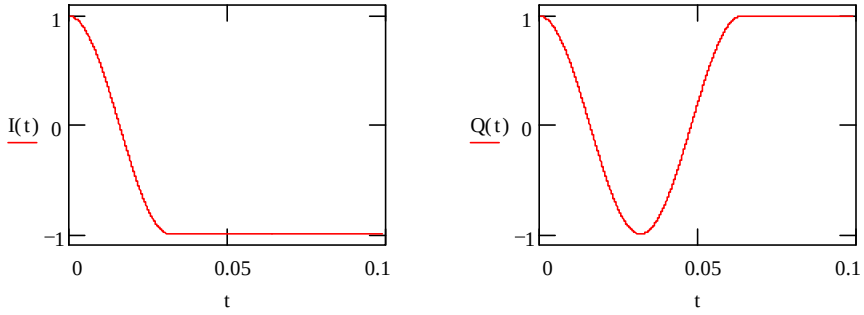


Note the gradual transition from one phase state to the next. This results in a much narrower bandwidth signal without the need for any post filtering. Also it can be seen that the amplitude of the signal is not constant. This means that the transmitter must not compress or limit the audio waveform otherwise the signal will again get much wider in bandwidth. This is perhaps the biggest problem with setting up a PSK31 station. It is very easy to overdrive and distort the PSK31 signal by applying the relatively high amplitude audio signal from the PC soundcard into the low level microphone input of a SSB transmitter. There is no easy way to monitor ones own signal for purity so one must rely on other's signal reports.

Finally, here is a MathCad simulation showing a QPSK signal that changes 180 degrees then by -90 degrees.

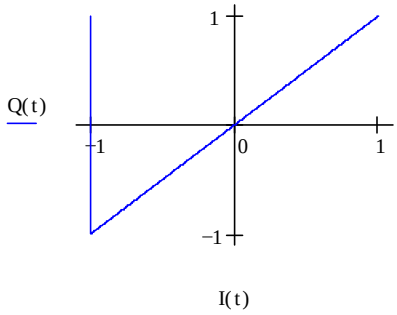
Modulation equations

$$I(t) := \text{if}\left(t<.032, \cos\left(\pi\cdot\frac{t}{T}\right), -1\right) \qquad Q(t) := \text{if}\left(t<.064, \cos\left(\pi\cdot\frac{t}{T}\right), 1\right)$$



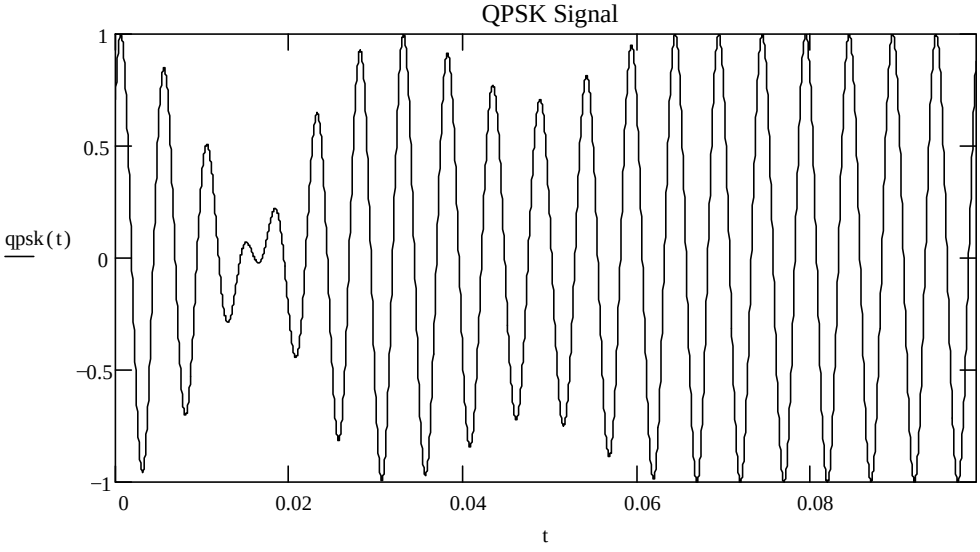
This is a 180 degree phase shift followed by a –90 degree phase shift.

QPSK Vectors



$$qpsk(t) := I(t)\cdot Ic(t) + Q(t)\cdot Qc(t)$$

QPSK Signal



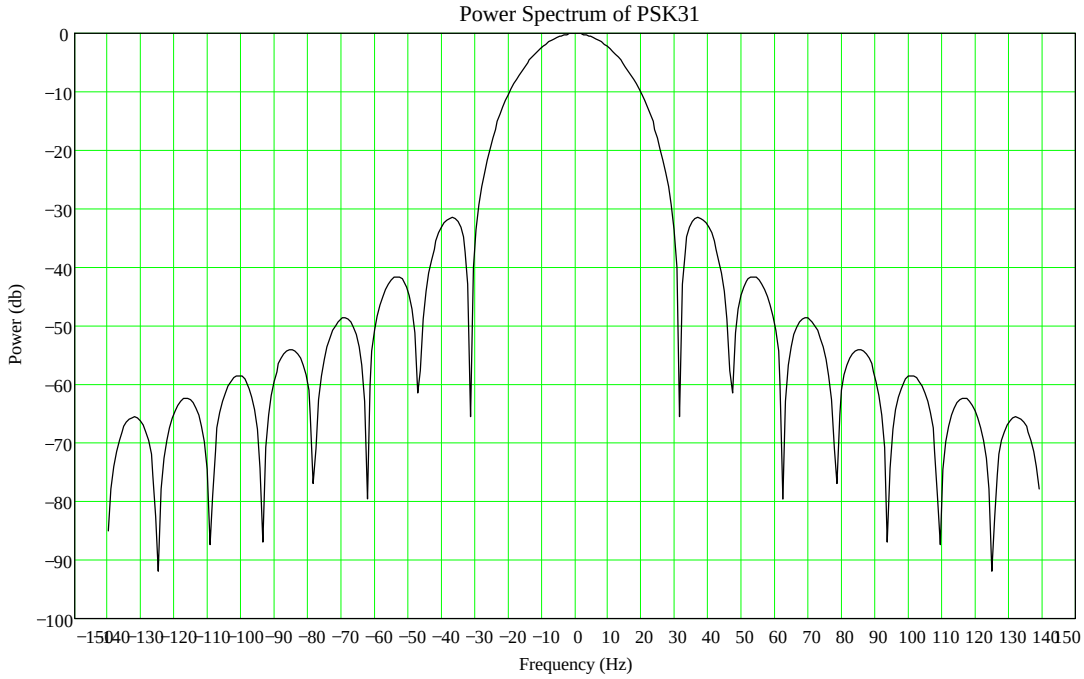
Note that during 90 degree phase changes, the amplitude does not drop all the way to zero as in the 180 degree case.

3.1.7 Power Spectrum

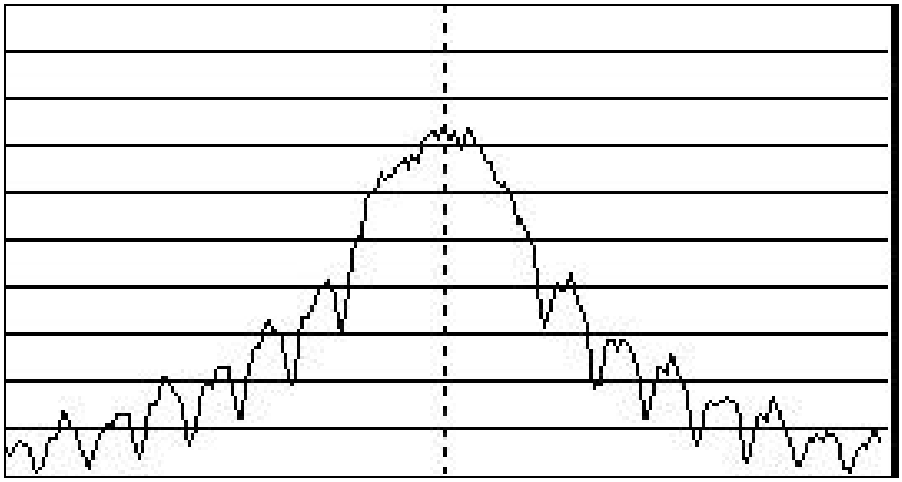
The BPSK/QPSK signal has a power spectrum consisting of a large main lobe centered around the carrier frequency out to a null at the carrier frequency +/- 31.25 Hz. There are multiple lobes extending out to infinity but their amplitudes continue to drop. Here is a MathCad simulation of the PSK31 power spectrum.

$$Fs := 31.25 \qquad T := \frac{1}{Fs} \qquad f := -140, -139.2 .. 139.2$$

$$S(f) := T \cdot \left(\frac{\sin(2 \cdot \pi \cdot f \cdot T)}{2 \cdot \pi \cdot f \cdot T} \right)^2 \cdot \left[\frac{1}{1 - 4 \cdot (f \cdot T)^2} \right]^2 \qquad Sdb(f) := 10 \cdot \log(S(f)) + 14.9$$



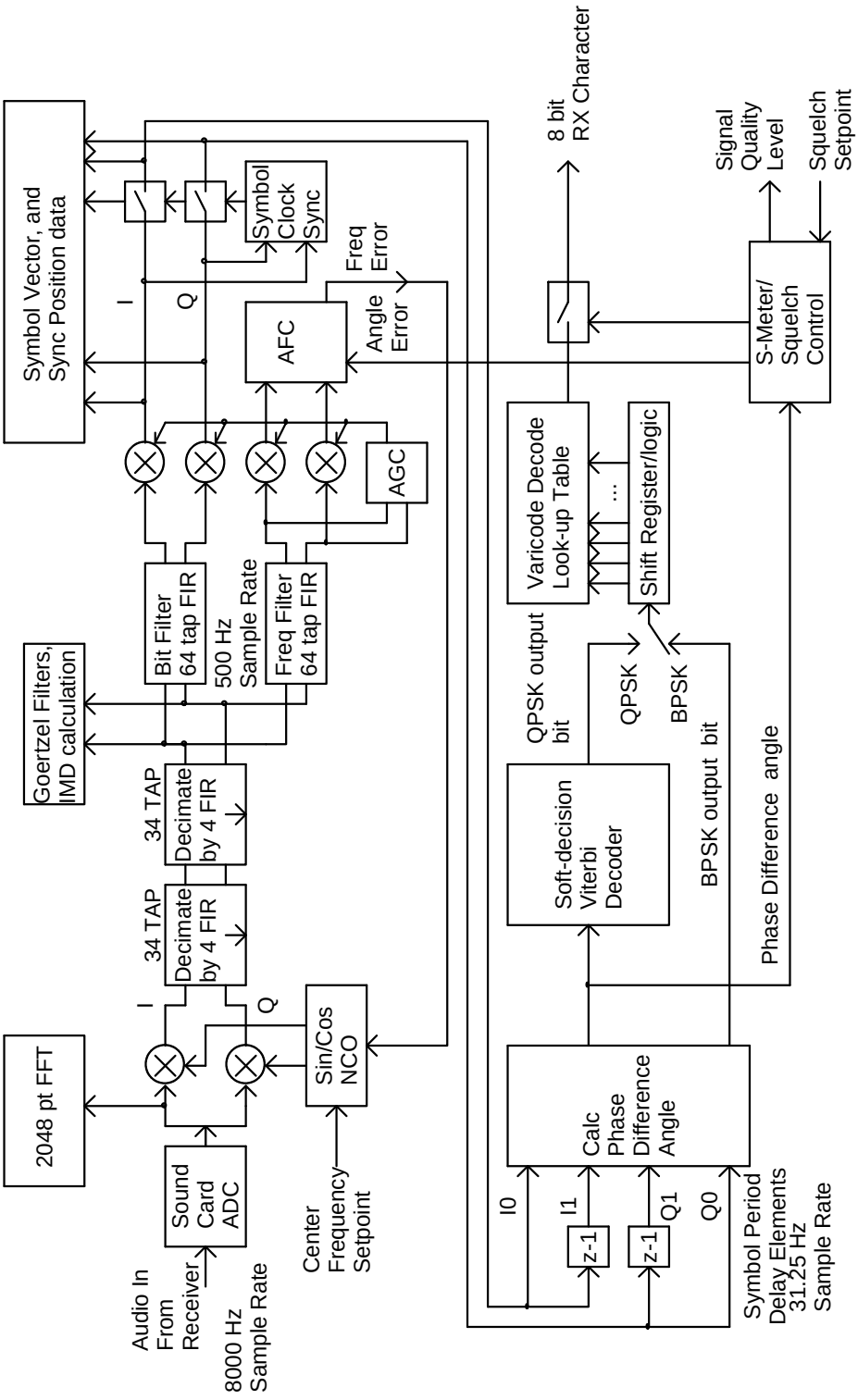
The following FFT scan of a QPSK signal compares favorably with the math model. The vertical divisions are 10 db.



3.2. PSK31 Signal Detection

3.2.1 Block Diagram

The following block diagram shows the major functions implemented by the dll to receive PSK31 signals. Receiver audio is captured by the PC soundcard and processed into final ASCII characters for display.



3.2.2 Soundcard Input

The receiver audio is sampled by the soundcard into 16 bit samples at a 8000 Hz rate and converted into floating point representation for the remainder of the processing.

This real signal is fed to 2048 point FFT and realtime display section for tuning and visual signal monitoring. It is also sent on to the next stage of the PSK decoder.

3.2.3 Complex Mixer

The next stage converts the real audio input into a complex baseband signal centered around the users center frequency set point. An NCO(numerically controlled oscillator) is implemented as a sin(wt) and a cos(wt) frequency source where ω is a control input from the users center frequency setpoint and also the AFC control signal that is derived further downstream. These two frequencies are 90 degrees apart and are multiplied by the real input signal to create two data streams called I and Q.

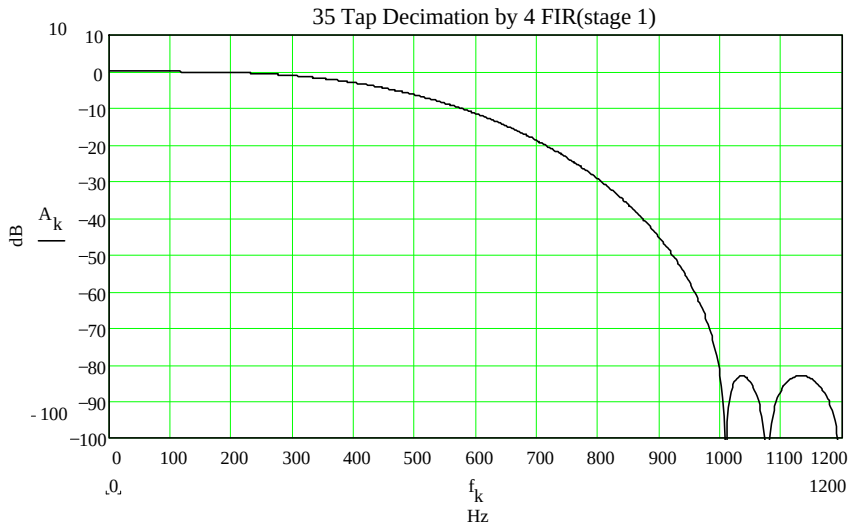
$$I(t) = Input(t)\cos(wt) \qquad Q(t) = Input(t)\sin(wt)$$

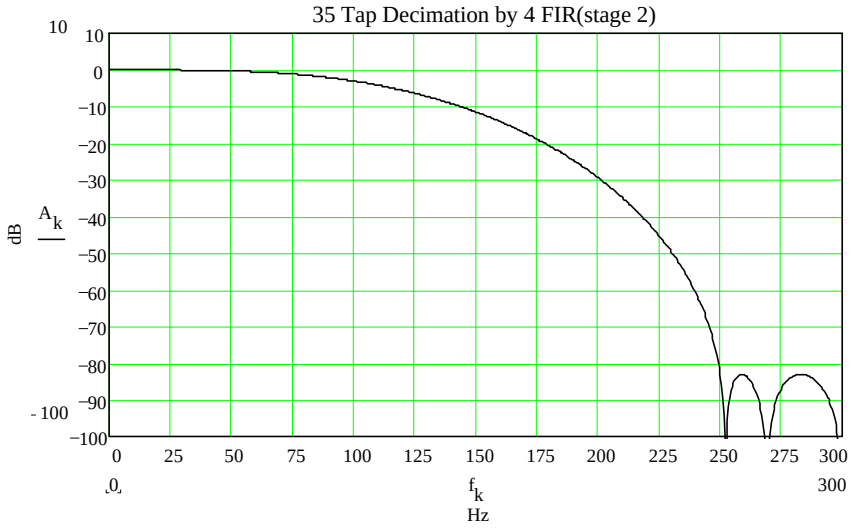
The following code segment performs this function:

```
//Generate complex sample by mixing input sample with NCO's sin/cos
      Inptr1->x = pIn[i] * cos( ncophz ); //generate I and Q signals
      Inptr1->y = pIn[i] * sin(ncophz);
      ncophz = ncophz + m_phzinc;         //update NCO
      if(ncophz > PI2)                    //handle 2 Pi wrap around
            ncophz -= PI2;
```

3.2.4 Decimation by 16

The complex signal is then down sampled by 16 to reduce the sampling rate to 16 times the signal bandwidth. Two stages of decimation by 4 are used rather than one. It is more processor efficient to break it up rather than have a fairly long FIR running at the highest sample rate. Each stage is identical and has a frequency response that is the same except that the one is scaled infrequency by 4. Since the signal is complex, the filters are run on both the I and Q signal.

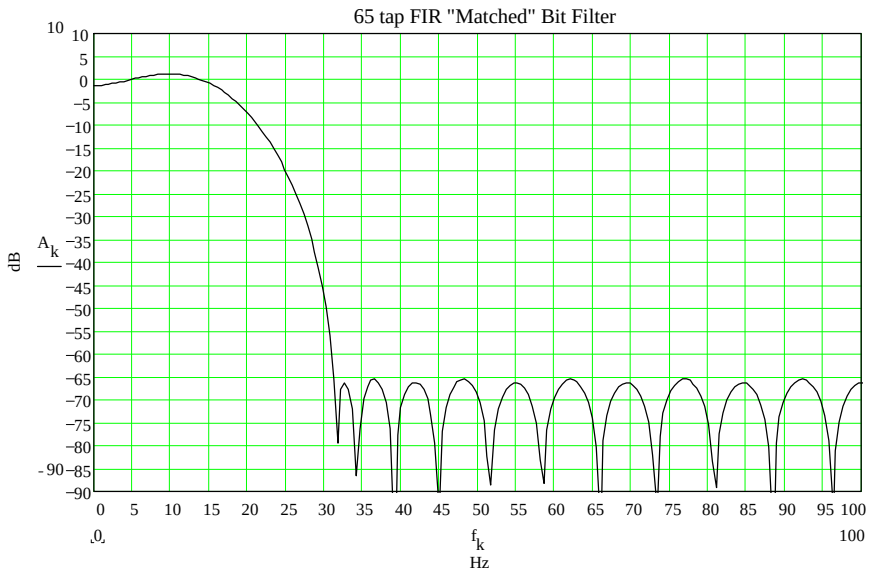




The final sampling frequency is now 88000/16 or 500 Hz.

3.2.5 Matched Data Bit filter

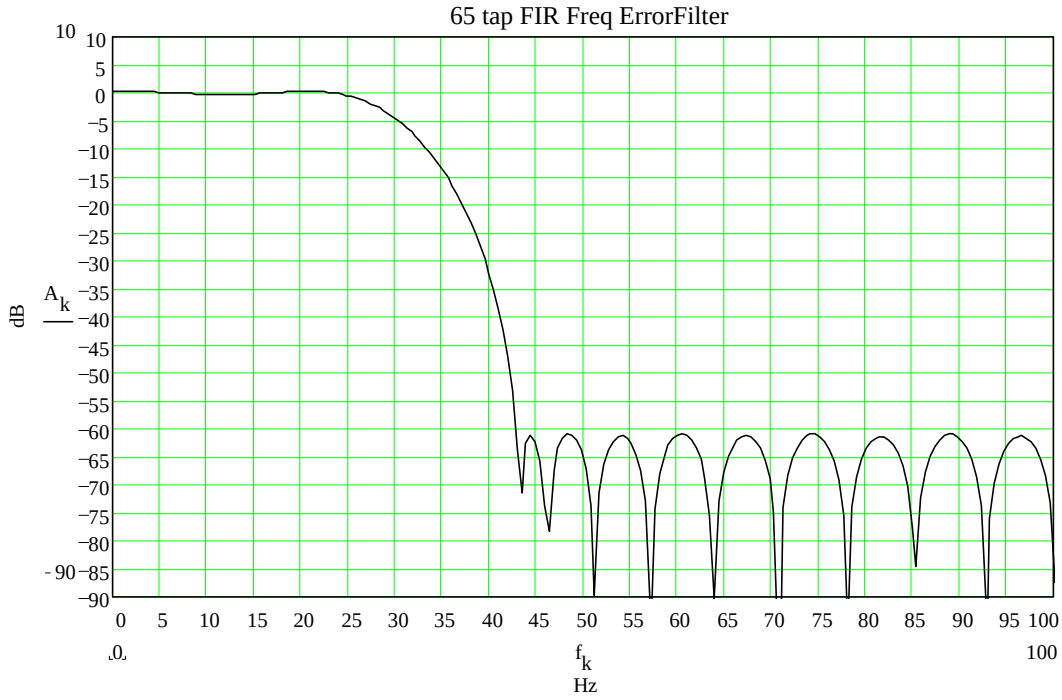
The final system bandwidth is set by this FIR filter. This filter has two purposes. One is to provide a magnitude response that provides the best signal to noise ratio in order to extract the data signal from the noise. The second thing it must do is minimize any ISI(Inter-Symbol Interference) that is generated in the transmitter, signal path, and receiver system. With PSK31 there is no ISI from the transmission process due to the wave shaping of the signal, so any ISI will come from the signal path and the receiver filters. Since the HF signal path is not predictable the best one can do is minimize the ISI generated by the receiver bit FIR filter. A compromise filter was developed that gives fairly low ISI and a reasonable cutoff shape. Better filters are probably out there but would not provide a whole lot of noticeable performance improvement, especially in the HF environment. The addition of interleaving or longer ECC codes would probably make a bigger difference. Below is the frequency response of the bit filter.



All the FIR filter coefficients were designed using either MathCAD or a program called PC-DSP by DSP Solutions.

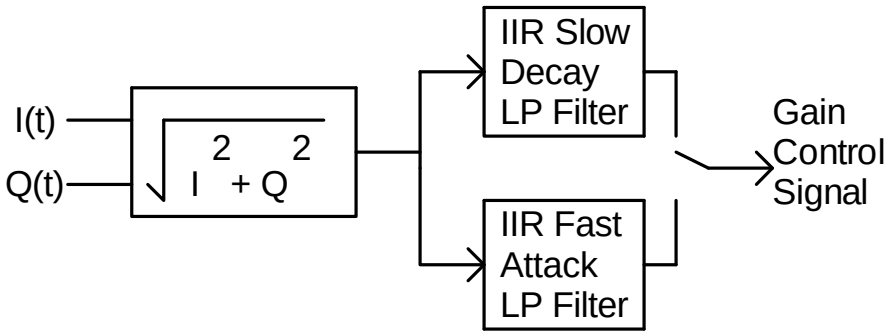
3.2.6 Frequency Error Filter

Unfortunately, the AFC(automatic frequency control) block could not use the output of the bit filter for locking on to the incoming signal frequency. The problem is that the bit filter is too narrow and the AFC can lock onto either side of the PSK31 idle signal which looks like two carriers spaced 15.625 Hz above and below the center frequency. The solution though wasteful was to use a separate filter just for the frequency control that was wide enough that it cannot distinguish between the PSK31 idle peaks. The response is only 6 dB down at 31.25 Hz so it spans both idle peaks.

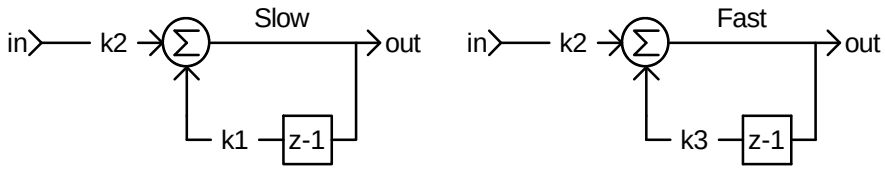


3.2.7 AGC

The AGC is derived from the average signal magnitude using the scheme shown below. The I and Q signals are then divided by the this AGC signal to help keep the average amplitude constant for the remainder of the processing.



Two different time constants are used depending on whether the signal is increasing in strength or decreasing. The low pass filters are simple IIR stages with one delay element. They have the same response as an analog RC filter. This type filter is useful for obtaining a very low cutoff frequency with little processor overhead. It can be implemented by one line of code. (y = k1*y + k2*x;) The down side is the poor frequency response as can be seen in the following plots.



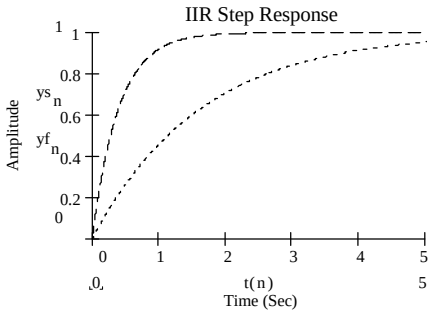
$t(n) := n \cdot \frac{16}{8000}$

$x_n := 1$

$KF := 200$ $KS := 816$

$k1 := 1 - \frac{1}{KS}$ $k2 := \frac{1}{KS}$ $k3 := 1 - \frac{1}{KF}$ $k4 := \frac{1}{KF}$

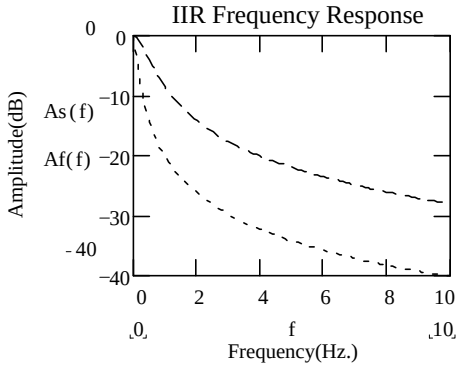
$ys_n := k1 \cdot ys_{n-1} + k2 \cdot x_n$ $yf_n := k3 \cdot yf_{n-1} + k4 \cdot x_n$



$Fs := \frac{8000}{16}$ $T := \frac{1}{Fs}$ $fs := 10$ $\delta f := 0.01 \cdot fs$

$f := 0, \delta f .. fs$ $z(f) := e^{(j \cdot 2 \cdot \pi \cdot f \cdot T)}$

$Hf(z) := \frac{k4 \cdot z}{z - k3}$ $Af(f) := 20 \cdot \log(|Hf(z(f))|)$ $Hs(z) := \frac{k2 \cdot z}{z - k1}$ $As(f) := 20 \cdot \log(|Hs(z(f))|)$



3.2.8 Frequency Error Detection/Correction

Tuning in a PSK31 signal is done in stages. When the fnSetRxFrequency(…) function is called, the following sequence of events takes place.

First, the FFT data is scanned within the capture range looking for the nearest signal peak. Normally this will get within a few Hz. of the PSK31 signal.

The second stage of frequency searching begins using the following wide range AFC algorithm for about 2.5 seconds.

The wide range AFC is performed by calculating the slope of the frequency within the frequency error filter bandwidth and essentially moving the NCO center frequency so that the frequency peak(if one exists) is at the center frequency. The phase of the signal is the arctan(I(t)/Q(t)). Since frequency is the derivative of phase, the signal frequency is just the derivative of the arctan function.
The following Mathcad simulation shows this relationship using two different frequency signals.

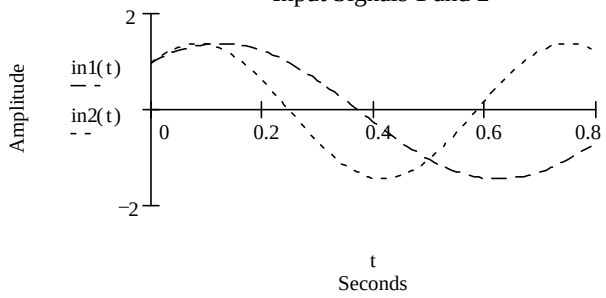
$F1 := 1$ $\quad$ $F2 := 1.5$

$I1(t) := \cos(2\cdot\pi\cdot F1\cdot t)$ $\quad$ $I2(t) := \cos(2\cdot\pi\cdot F2\cdot t)$

$Q1(t) := \sin(2\cdot\pi\cdot F1\cdot t)$ $\quad$ $Q2(t) := \sin(2\cdot\pi\cdot F2\cdot t)$

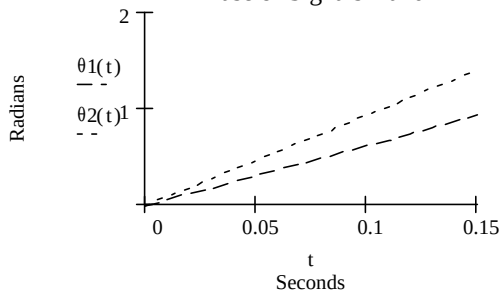
$in1(t) := I1(t) + Q1(t)$ $\quad$ $in2(t) := I2(t) + Q2(t)$

Input Signals 1 and 2

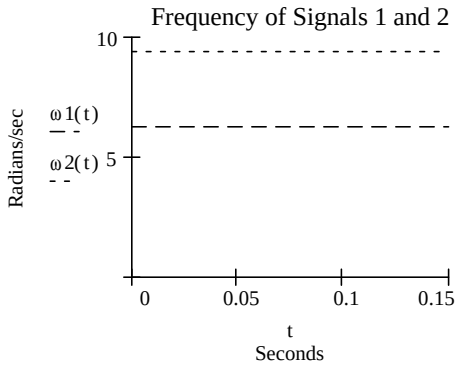


$\theta1(t) := \operatorname{atan}\left(\frac{Q1(t)}{I1(t)}\right)$ $\quad$ $\theta2(t) := \operatorname{atan}\left(\frac{Q2(t)}{I2(t)}\right)$

Phase of Signals 1 and 2



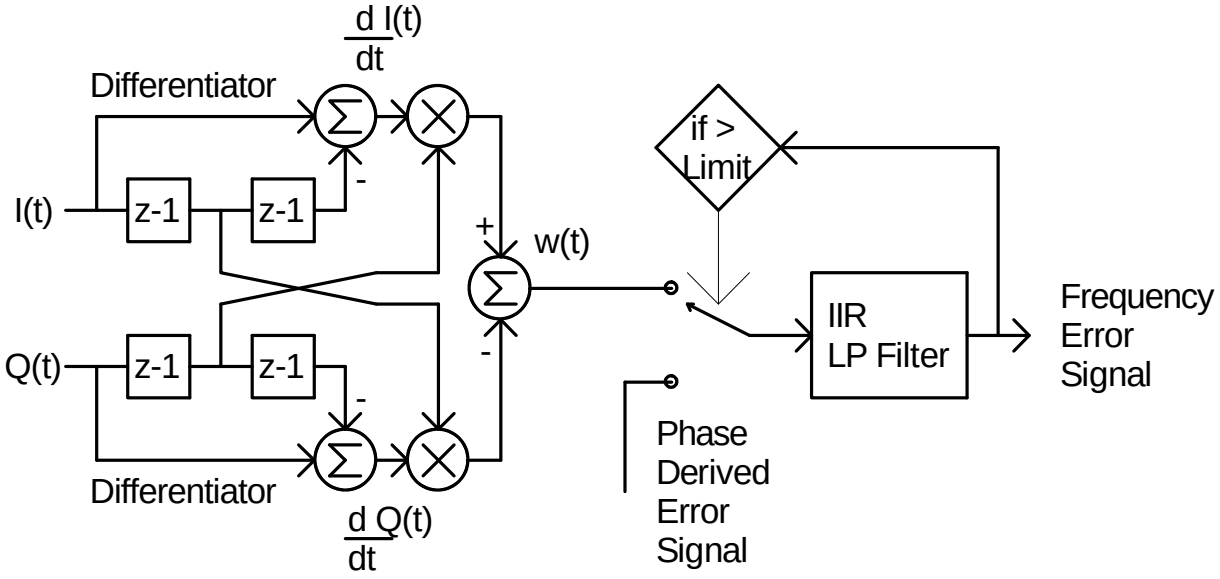
$\omega1(t) := \frac{d}{dt}\theta1(t)$ $\quad$ $\omega2(t) := \frac{d}{dt}\theta2(t)$



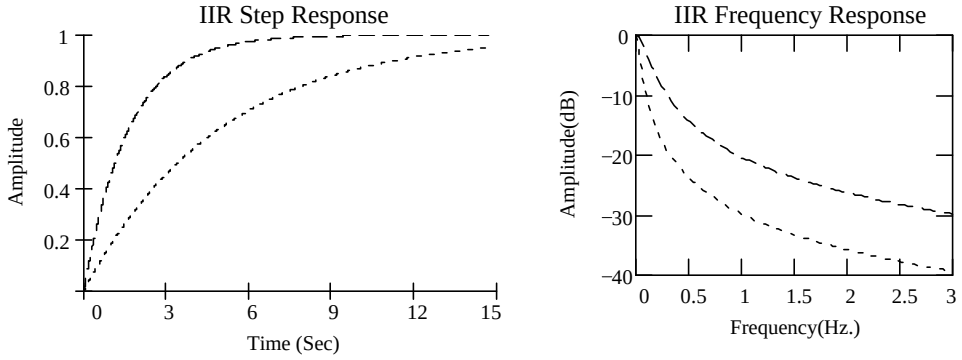
From a dusty calculus book the following identity was obtained that gives the signal frequency as a function of the I and Q signals and their derivatives without the use of the atan() function.

$$\frac{d}{dt}\operatorname{atan}\left(\frac{Q(t)}{I(t)}\right) = \frac{1}{1+\left(\frac{Q(t)}{I(t)}\right)^2}\cdot\frac{d}{dt}\frac{Q(t)}{I(t)} = \frac{I(t)\cdot\frac{d}{dt}Q(t) - Q(t)\cdot\frac{d}{dt}I(t)}{I(t)^2 + Q(t)^2}$$

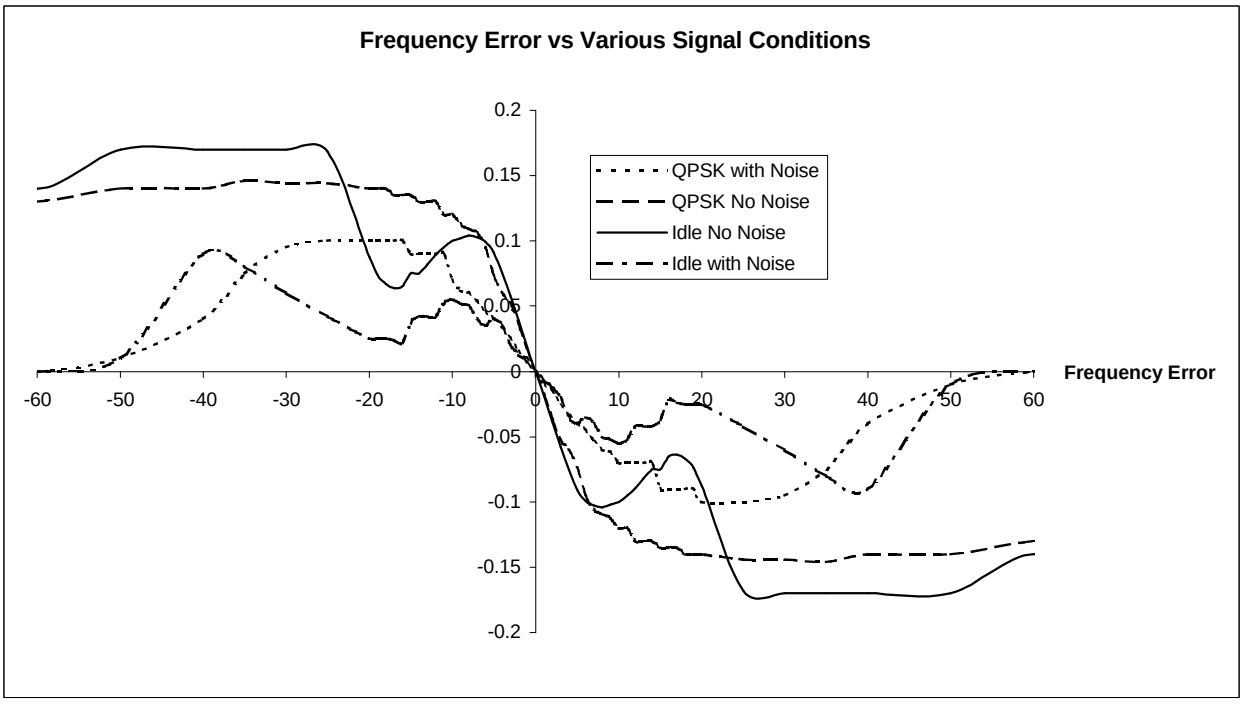
If the magnitude of the I/Q signal is adjusted to always be equal to one by the use of some AGC, then the denominator can be ignored. This allows the implementation of the frequency detector to be implemented using two differentiators as shown in the following:



The IIR LP filter are the same type as used in the AGC section with different time constants.



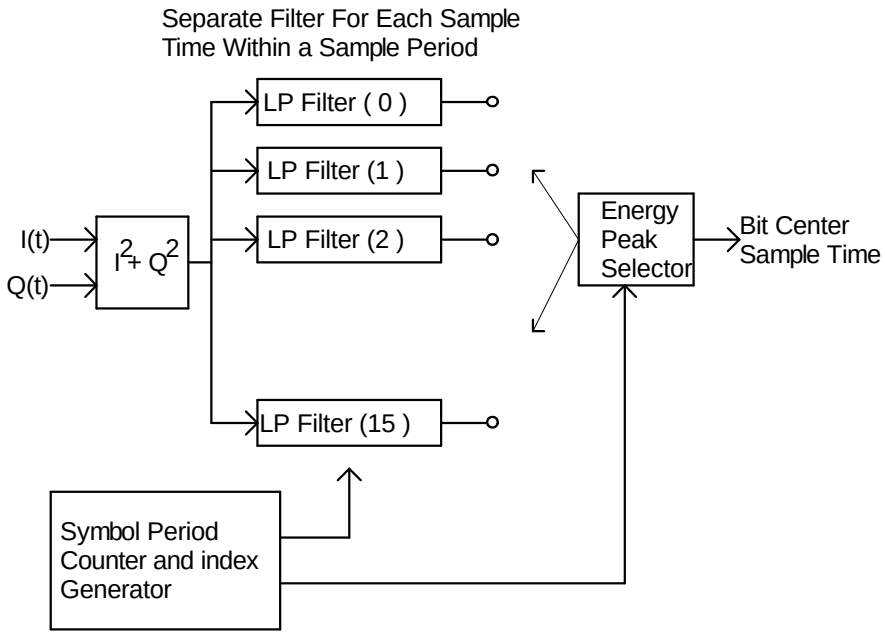
The transfer function for the differential frequency error block was obtained experimentally and is plotted below. It is interesting to note the dip in the function around +/- 16 Hz on the idle(180 deg. shifting) signal. This is due to the two frequency components of the idle signal. As long as the error signal doesn't change sign at this dip, the loop will still lock correctly at the center frequency. The slope of the transfer functions change due to the presence of noise. This is due to the AGC acting on the noise and reducing the actual signal level, which as was shown earlier, must remain relatively constant in order for this frequency detector scheme to work. The noise level plotted here is right at the threshold of signal detection, so is worst case.



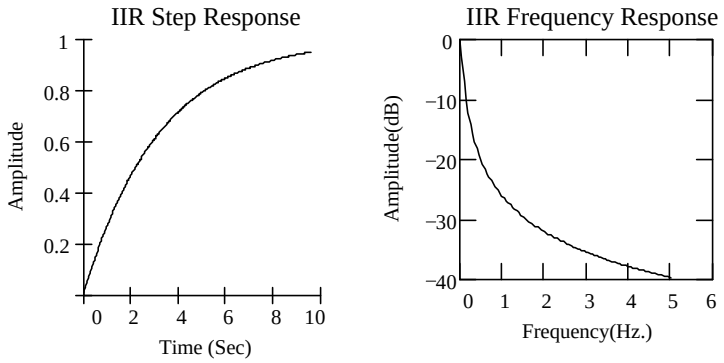
After the 2.5 second wide range AFC action a narrow and slow acting phase derived error signal is used. This phase error signal is derived from the difference angle of the baseband signal (described later). This error signal is generated by how far from the ideal 0 or π phase position the signal is. If the frequency is off, then the phase difference of the PSK31 signal will be rotated from the ideal position and an error metric can be derived. This method can only be used for a very narrow range of a few Hz and so is only useful once the main frequency error has dropped within 3 Hz in QPSK mode and 5 Hz in BPSK mode. This narrow mode AFC remains in operation until the user changes frequency again.

3.2.9 Symbol Synchronization

The next level of synchronization is to find the center of each symbol in order to sample it at the optimum time. Several schemes were tried with varying success. The classic early-late synchronizer was tried that integrates the signal energy over part of the symbol time and then again with a small time delay. An error signal can be obtained that is fed back to adjust a symbol clock. This works but had problems with noisy QPSK signals. Another method was tried using an algorithm that selectively finds the peaks and valleys in each I and Q signal. This method worked OK but was complicated. As a side benefit it could provide a good signal quality metric that worked well with very noisy signals. However, the final method chosen was a simple method that seems to work quickly and well is shown by the following diagram.

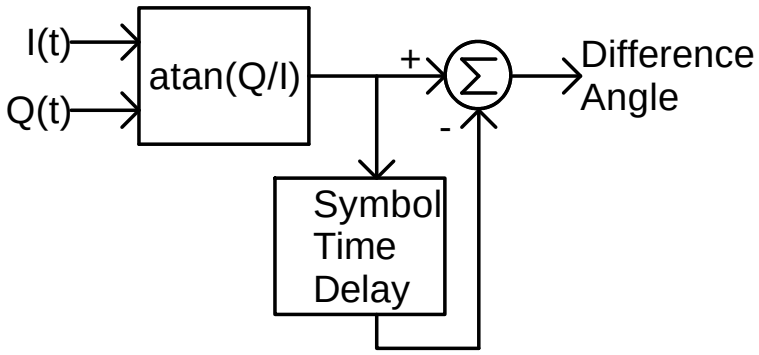


There are 16 samples per symbol at the 500 Hz. Sample rate. The energy in the input signal at each sample time is individually filtered and stored in a filter array. At each symbol period of .032 seconds, the filter that has the most energy is selected and the sample point associated with this sample is assumed to be the center of the data symbol. The LP filters are again the simple IIR's with the following characteristics:



3.2.10 Squelch Function

To implement a squelch function, a measure of signal quality is needed. Visually, one can see how good the signal is by noticing how the incoming signal vectors are distributed. A strong signal has most of the vectors tightly distributed around the perpendicular axis. A noisy signal will have a wide distribution around the axis. By creating a histogram of the incoming signal angles and looking at the average deviation that they take from the "ideal" 0, 90, -90, and 180 positions, one can extract a signal proportional to signal quality. In order to use the same algorithm for BPSK and QPSK, only the vectors around 0 and 180 degrees are analyzed.
First, the incoming signal difference angle must be found. The direct approach would be to just take the arctan(Q/I) and subtract the previous arctan(Q/I) from it to get the difference angle.



This method has a couple of problems. One is that if I(t) is zero things blow up. Second, there must be extra logic to do the subtraction since the atan function doesn't return a nice 0 to 360 degree range. One must keep track of which quadrant the vectors are in and subtract accordingly.
A different approach is used that creates a third vector whose angle is the difference angle but does not use the atan function. This vector is created geometrically and so does not suffer from the discontinuities of the transcendental atan() function. The atan function can now be used on this new vector to find it's angle directly. The atan function can still blow up but only if both I and Q are zero which is less likely to occur.
To create this third difference vector one simply multiplies the current sample I,Q vector by the complex conjugate of the previous I,Q vector.

$$Y_k = A_k \cdot e^{j\cdot\theta_k} \qquad Y_{k-1} = A_{k-1} \cdot e^{j\cdot\theta_{k-1}} \qquad \left(\overline{Y}\right)_{k-1} = A_{k-1} \cdot e^{-j\cdot\theta_{k-1}}$$

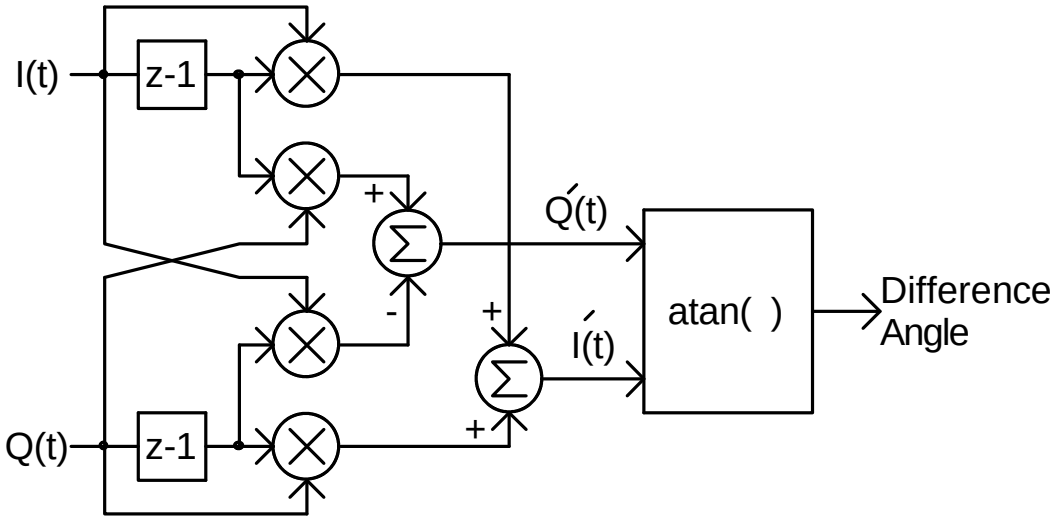
$$Z_k = Y_k \cdot \left(\overline{Y}\right)_{k-1} = A_k \cdot A_{k-1} \cdot e^{j\left(\theta_k - \theta_{k-1}\right)}$$

Z_k = Vector whose angle is the difference between the original vectors.

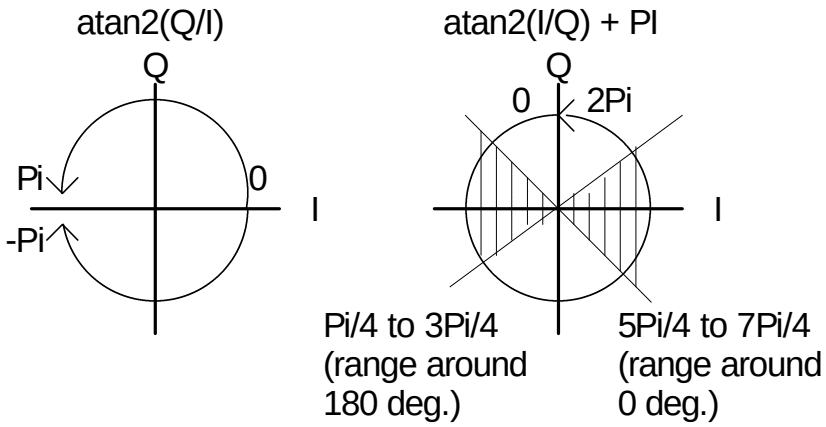
In terms of the complex coordinates I and Q,

$$Z_k = Y_k \cdot \left(\overline{Y}\right)_{k-1} = \left(I_k + jQ_k\right) \cdot \left(I_{k-1} - jQ_{k-1}\right)$$
$$= \left(I_k \cdot I_{k-1} + Q_k \cdot Q_{k-1}\right) + j\left(Q_k \cdot I_{k-1} - I_k \cdot Q_{k-1}\right)$$

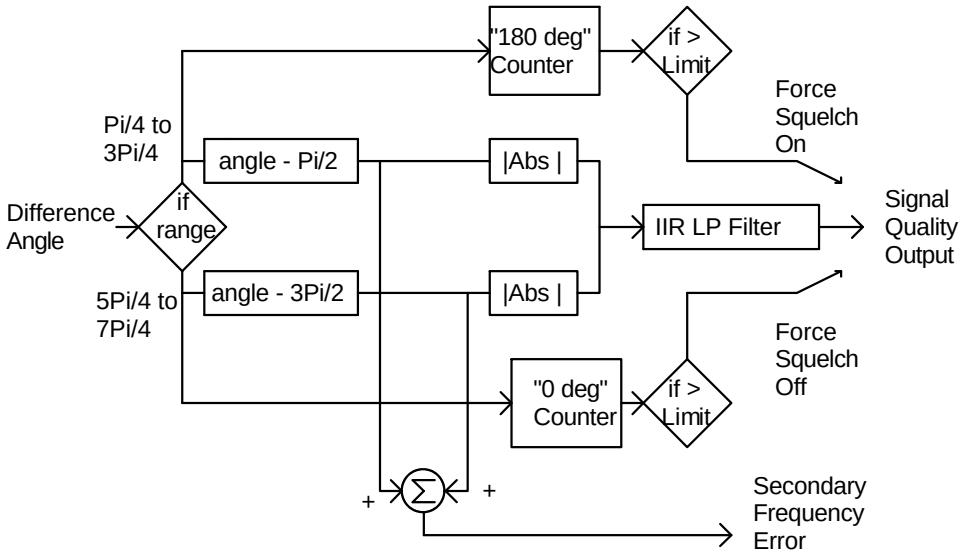
In diagram form this implementation is show below.



In actuality, a 'C' function atan2(y,x) is used to obtain the angle. It returns a value from –PI to +PI. This still causes some grief if one wants to create a histogram of angles around 0 and around PI(0 and 180deg). If one swaps I and Q, the affect is to map 0 and 180 degrees to +/- 90 degrees. This translates the two ranges of interest(-PI/4 to PI/4 and 3PI/4 to 5PI/4), into the new ranges of PI/4 to 3PI/4, and 5PI/4 to 7PI/4.



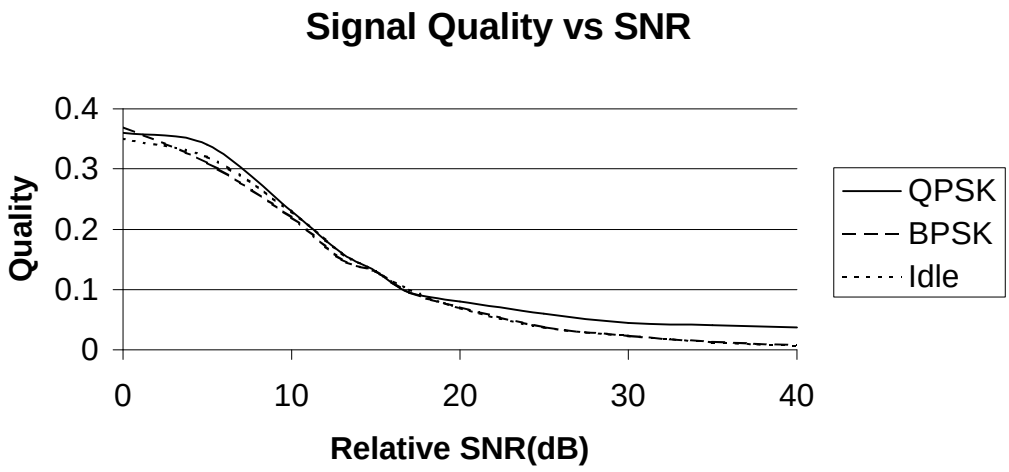
The basic signal quality scheme is then implemented as follows:



If the incoming difference angle is in the "ZERO" range, the absolute value of the angle and Pi/2 is found. If the incoming difference angle is in the "180 degree" range, then the absolute value of the angle and 3Pi/2 is found. The two values are added together and run through a low pass filter. This is a measure of how far away from the ideal the angle is, and is used for squelch control and signal quality display.

Peter Martinez specified a feature into the PSK31 signal scheme in which each transmission should begin with a string of at least 32 consecutive 180 degree shifting "idle" symbols and also each transmission should end with a string of 32 consecutive 0 degree non-shifting symbols(steady carrier). The reason was twofold. The beginning idle string gives the decoders a chance at synchronization before any data is sent. It can also be used for squelch functions to indicate a transmission is starting. The trailing carrier can also be used to deactivate the squelch since a string of 32 "0 degree" symbols cannot occur during any data transmission.

Two counters are used to count consecutive idle characters and solid carrier symbols. If either reaches it's limit, it bypasses the normal slow acting signal quality signal and forces the squelch either on or off. This gives a much quicker acting squelch under good signal conditions.



Note how the QPSK signal with very high SNR still has a lower quality signal. This is due to ISI from the bit filter giving the signal some phase jitter.

This block is also used to derive the secondary frequency error signal that is used along with the main differential frequency error generator to lock onto the center frequency. The error signal is taken from the signal quality block before the absolute value function and IIR filter. This signal then has the sign and magnitude information that can be used to nudge the main mixer NCO toward the true center frequency. This error signal kicks in when the overall frequency error is less than 3 Hz.

3.2.11 IMD Measurement

One of the problems with PSK31 transmission is that most amateurs do not have the test equipment needed to measure the distortion in their transmitted signals. A spectrum analyzer is required to accurately measure the distortion products present in the transmitted signal. Currently one must rely on received signal reports to determine the quality of their signals. One method used is to calculate the IMD(InterModulation Distortion) on the received signal. Although not the ideal way, this method can be useful if one is aware of the limitations to this method.

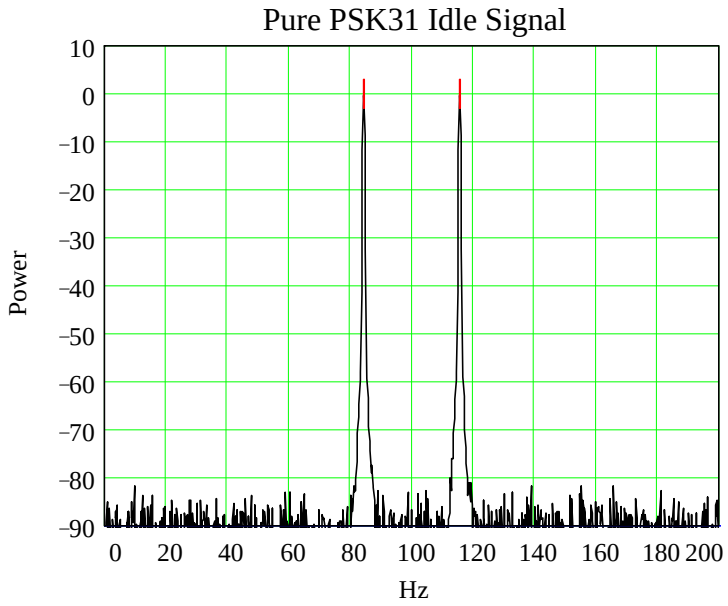
3.2.11.1 Measurement Method

One method of IMD measurement involves using two non-harmonically related tones to modulate the transmitter and observing the output for any spurious tones other than the two being used. If any non-linear amplification occurs in the transmitter, spurious frequencies will be generated that are combinations of the original tones and multiples of the sum and difference of the original tones.
If,
F1 = tone 1
F2 = tone 2
then the following frequencies can be generated:

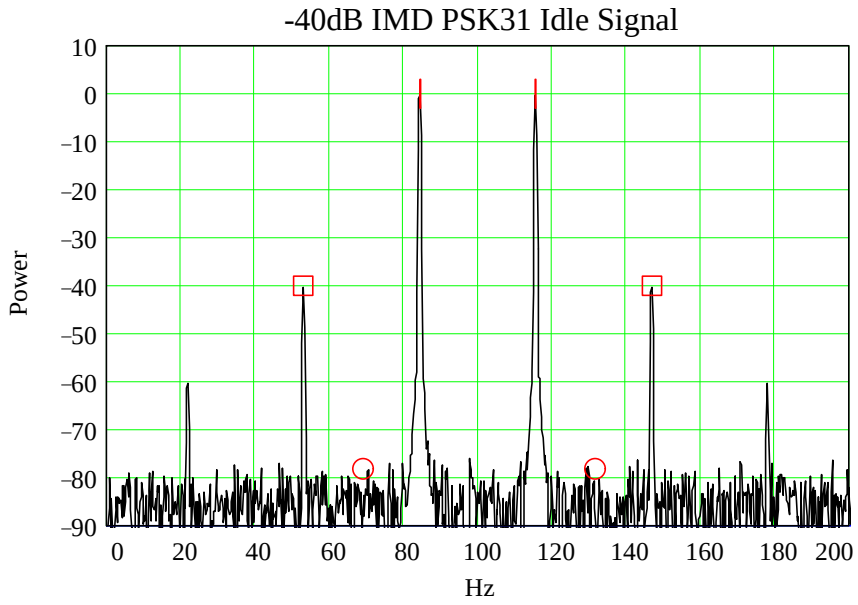
+/-nF1 +/- mF2 where n and m are integers.

The most common and strongest distortion product is called the third order product and is the frequencies of (2F1 – F2) and (2F2-F1).

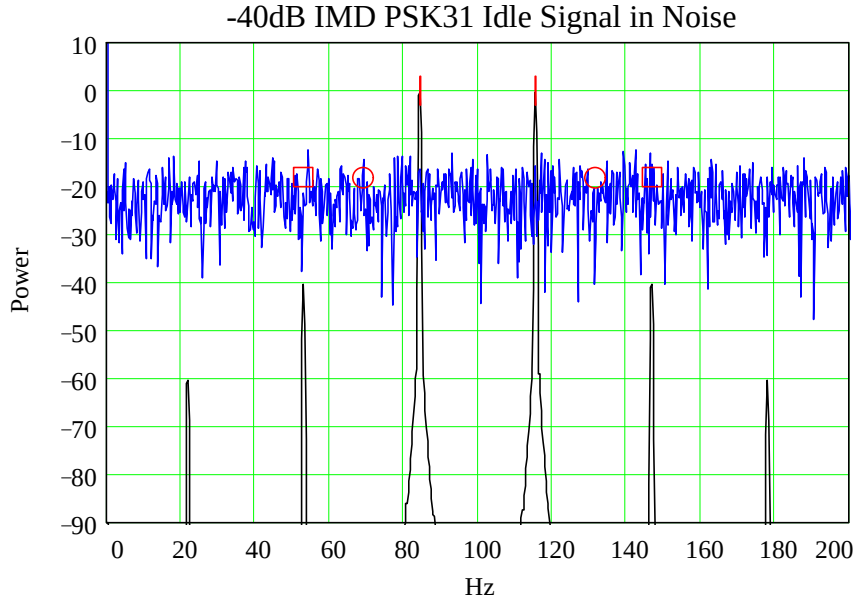
If one examines the PSK31 idle signal, it can be shown that the constant phase change of 180 degrees and the cosine shaped envelope of the signal generates two tones that are at the PSK31 center frequency +/- 15.625Hz. These tones can be used to measure the IMD of a transmitter.
The following shows a PSK31 idle signal with a center frequency of 100Hz. The two tones are at 100+/- 15.625Hz or 84.375Hz and 115.625Hz.



The third order product frequencies are (2F1 – F2) and (2F2-F1) or 53.125Hz and 146.875Hz. The following shows a PSK31 idle signal with some 3rd (and 5th) order IMD.
By measuring the power difference between the original tones and the 3rd order IMD tones, one can provide a measurement as to how much distortion is on the signal. In this case, the IMD is –40dB since the original tones are at 0dB and the 3rd order tones are at –40dB.



This works well with a strong received signal but look what happens if the signal is weak in the presence of noise.



Note that if one measures the signal level at the 3rd order frequencies, then erroneous results are obtained because the noise level is higher than the distortion product. An IMD reading of –20dB will be obtained even thought the signal is actually –40dB.
This is why it is important that the received signal be well above the noise floor before trying to obtain an IMD reading. Another factor is the HF propagation effects that also introduce errors into this measurement method.

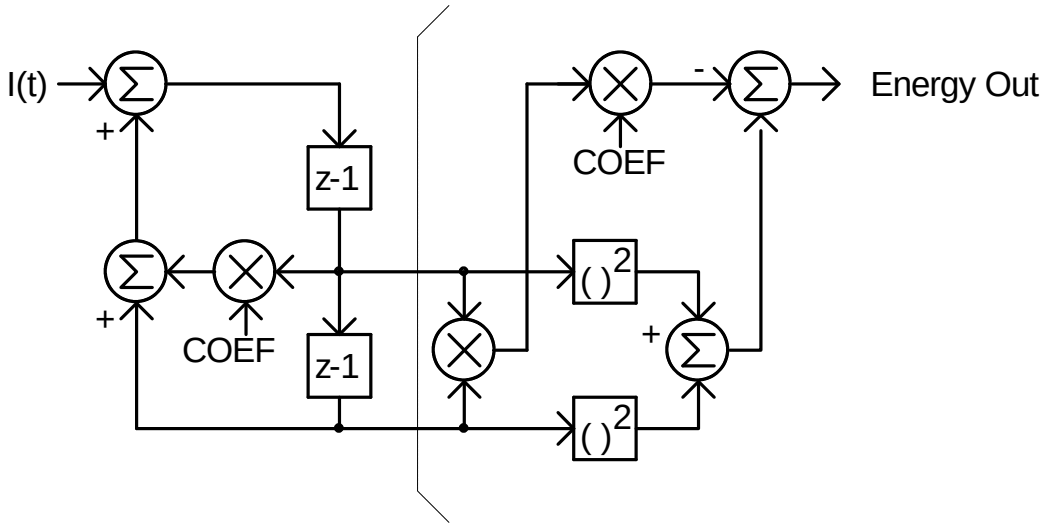
The PSKCore algorithm samples the original PSK31 idle tones, the 3rd order distortion tones, as well as the noise floor level. If the noise floor level is higher than the IMD tone then the IMD measurement is flagged as suspect.(See description of the MSG_IMDRDY message)

3.2.11.2 Goertzel Filter

The PSKCore dll uses Goertzel tone filters to sample the incoming signal and measure the IMD present. Goertzel filters are essentially IIR filters that calculate discrete Fourier transform frequencies. If a small number of frequencies are needed, this is much more efficient than calculating the entire fft. These filters are commonly used in DTMF tone detection since only 8 tones are required. Another advantage of the Goertzel algorithm is that only part of the filter needs to be calculated every sample time so a further savings is obtained.
A block diagram of the modified Goertzel algoritm is shown below. The left side is calculated every sample time while the right side energy calculation only needs to be done every N samples where N is the Goertzel filter length. The constant "COEF" and value N determine the frequency and bandwidth of the tone detector.

COEF = $2\cos(2\pi k/N)$ where k is the nearest integer that satisfies the equation:

k = N fi/fs where N is the filter length, fi is the desired filter frequency, and fs is the sample frequency.



For the PSKCore dll, the following parameters are used for the IMD measuring filters:
N = 288
fs = 500Hz
The signal is measured after the complex mixer so the signal center frequency is zero.
f0 = 15.625 Hz this is the PSK31 idle tone frequency
f1 = 31.25 Hz this is the frequency for measuring the noise floor
f2 = 46.875 Hz this is the 3rd order IMD frequency

k0 = Nf0/fS = 9	COEF_0 = $2\cos(2\pi k0/N)$ = 1.96
k1 = Nf1/fS = 18	COEF_1 = $2\cos(2\pi k1/N)$ = 1.8477
k2 = Nf2/fS = 27	COEF_2 = $2\cos(2\pi k2/N)$ = 1.663

The basic algorithm is implemented as follows:

```
temp = I1;
I1 = I1*COEF-I2+samp;
I2 = temp;
if( ++NCount >= N )
{
      NCount = 0;
      Energy = I1*I1 + I2*I2 - I1*I2*COEF;
      I1 = I2 = 0.0;
}
```

If a constant string of PSK31 idle symbols has been received then the following code is executed to calculate the IMD of the incoming signal:

```
////////////////////////////////////////////////////////////
//  This routine calculates the energy in the frequency bands of
//   carrier=F0(15.625), noise=F1(31.25), and
//   3rd order product=F2(46.875)
////////////////////////////////////////////////////////////
BOOL CCalcIMD::CalcIMDValue( INT &imdval)
{
        m_Snr = 10.0*log10(m_Energy[0]/m_Energy[1]);
        m_Imd = 10.0*log10(m_Energy[2]/m_Energy[0]);
        imdval = (INT)m_Imd;
        if( m_Snr > (-m_Imd+6) )
                return TRUE;
        else
                return FALSE;
}
```

3.2.12 Symbol Decoding

The next step is to convert the I and Q signals back into the four possible symbols (two for BPSK). PSKCore finds the difference angle as described in the squelch section and find the nearest 0, 90, -90, or 180 degree position and that is the symbol to use in the decoder.

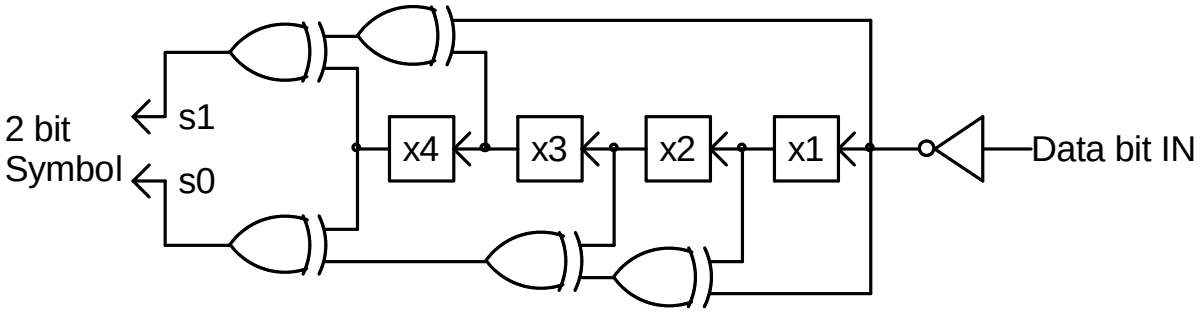
3.2.12.1 BPSK Decoder

Decoding the BPSK signal is fairly straight forward. If the signal phase difference angle is less than 90 degrees then the bit is a One. If it is greater than 90 degrees than the bit is a Zero. This bit steam is shifted into a shift register until an inter-character marker of 2 or more Zeros is found. The shift register value is then used as an index into a reverse Varicode table to obtain the original character. There is no error correcting with this mode.

3.2.12.2 Soft Viterbi Decoder

The Viterbi Decoder tries to reconstruct the original transmitted signal by looking at the sequence of received signals and comparing it to all the possible transmitted sequences. The sequence(or path) that has the best match is chosen to provide the best guess at a current data bit.
The process is similar to being lost in your car and you try to find out where you are on a map by observing how far you have traveled, which way you have turned, etc. By looking at all possible roads on the map with the same turns and distances that you've taken, you pick the roads that best match your route and conclude where you are. Viterbi added a simplifying step where if two paths end up at the same intersection, you pick the path with the best match at that point, and eliminate the other. The paths that are left(survivors) are the only ones left in contention for future calculations.
The details of the Viterbi algorithm can be found in most communications books so it won't be discussed in much detail here.

PSKCore uses a modified implementation of a Viterbi decoder described in an article by Peter Martinez. Soft decision capability was added and the path metrics converted from integer to floating point representation.
Recall how the transmitted data is encoded using the following state machine. Note that 4 memory stages plus the current data bit is used to form the output symbol. Every input bit causes the state machine to transition to one of 16 possible states. The 2 bit output symbol is derived from the state and the input bit.



The Viterbi decoder uses the same encoder to try all possible transmitted combinations and calculate an error metric based on how "far away" from each possibility the received symbol is.
The algorithm is executed at every symbol period with the phase angle as input into the decoder routine. The algorithm begins by filling two temporary 32 element arrays in the following manner. The index into the arrays is all possibilities that the constraint length 5 encoder can have.(2^5=32). The distance(how far away from +90,-90,0,or 180) is calculated for all possibilities.
For each possible index, a new path distance is computed by adding the existing survivor path distance and the new symbol's error distance. A second array (bitestimates[]) gets the new bit pattern estimate from the existing survivor paths plus the new bit combination being examined. The minimum of all the path distances is kept for use later normalization.

```
//////////////////////////////////////////////////////////////////////
// Soft-decision Viterbi decoder function.
//////////////////////////////////////////////////////////////////////
BOOL CPSKDet::ViterbiDecode( double newangle)
{
double pathdist[32];
double min;
INT bitestimates[32];
INT ones;
INT i;
const double* pAngleTbl;
        min = 1.0e100;                  // make sure can find a minimum value
if( newangle >= PI2/2 )                 //deal with ambiguity at +/- 2PI
        pAngleTbl = ANGLE_TBL2;         // by using two different tables
else
        pAngleTbl = ANGLE_TBL1;
for(i = 0; i < 32; i++)                 // calculate all possible distances
{                                                                       //lsb of 'i' is newest bit estimate
        pathdist[i] = m_SurvivorStates[i / 2].Pathdistance +
                        fabs(newangle - pAngleTbl[ ConvolutionCodeTable[i] ]);
        if(pathdist[i] < min)   // keep track of minimum distance
                min = pathdist[i];
        // shift in newest bit estimates
        bitestimates[i] = ((m_SurvivorStates[i / 2].BitEstimates) << 1) + (i & 1);
}
}
```

The next step is to index through all the path distances and eliminate all the paths that reached the same state but have higher path distances. The shortest path is copied into the survivor state array along with the associated bit pattern estimate. Note also that the minimum value calculated earlier is subtracted from the total path distance before being saved into the survivor array. This keeps the value from growing over time and remain bounded.

```
for(i = 0; i < 16; i++) //compare path lengths with the same end state
                                // and keep only the smallest path in m_SurvivorStates[].
{
        if(pathdist[i] < pathdist[16 + i])
        {
                m_SurvivorStates[i].Pathdistance = pathdist[i] - min;
                m_SurvivorStates[i].BitEstimates = bitestimates[i];
        }
        else
        {
                m_SurvivorStates[i].Pathdistance = pathdist[16 + i] - min;
                m_SurvivorStates[i].BitEstimates = bitestimates[16 + i];
        }
}
```

Finally the survivor state array bit estimates are examined 20 bits back in time, and the majority of ones or zeros is used as the best estimate for the transmitted bit. If there is a tie, a fair "coin toss" is used to guess at the bit. It has been shown that calculating over 4 or 5 times the constraint length does not significantly improve performance of the Viterbi decoder.

```
ones = 0;
for(i = 0; i < 16; i++)              // find if more ones than zeros at bit 20 position
        ones += (m_SurvivorStates[i].BitEstimates&(1L << 20));
if( ones == (8L << 20 ) )
        return ( rand() & 0x1000 );     //if a tie then guess
else
        return(ones > (8L << 20) );     //else return most likely bit value
```

This bit steam is shifted into a shift register until an inter-character marker of 2 or more Zeros is found. The shift register value is then used as an index into a reverse Varicode table to obtain the original character.

4. Further References and Acknowledgments

The following are some references used in this document and can be used for further reading on the subject.

- Peter Martinez G3PLX. "PSK31: A new radio-teletype mode with a traditional philosophy"
- Peter Martinez G3PLX. "PSK31 Fundamentals"
- MathCad ver.6.0. MathSoft 101 Main St.,Cambridge, MA 02142
- Marvin E. Frerking. "Digital Signal Processing in Communication Systems"p.444. ISBN0-442-01616-6
- George B. Thomas, Jr. "Calculus and Analytic Geometry" p.238
- W.T. Webb and L. Hanzo "Modern Quadrature Amplitude Modulation" p.367 ISBN0-7273-1701-6
- Yuri Okunev. "Phase and Phase Difference Modulation in Digital Communications" p.173-216 ISBN 0-89006-937-9
- Bernard Sklar. "Digital Communications Fundamentals and Applications" ISBN 0-13-211939-0
- "C. Britton Rorabaugh. "Error Coding Cookbook". P. 127 ISBN 0-07-911720-1
- Tom McDermott, N5EG. "Wireless Digital Communications: Design and Theory" ISBN 0-9644707-2-1
- Peter Martinez G3PLX. "Description of the Half-Rate QPSK code proposed for the QPSK/FEC Extension to PSK31"

I would like to thank Bob, K4CY for suggestions and helping find bugs in the DLL and also Julian, G4ILO for providing the Delphi function prototyping.