WinPSK Technical Reference Manual

by

Moe Wheatley, AE4JY
ae4jy@mindspring.com

Table of Contents

1. WinPSK Overview

1.1. Introduction

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

This document was written to describe some of the internal workings of the WinPSK program that was developed as a result of my experimenting with DSP on a PC soundcard. Previously, experimenting with DSP was achieved using evaluation boards from various DSP chip manufacturers. Programming these boards was tedious due to their assembly language and fixed-point number representation. Trying to learn the basics of DSP often got lost in the details of programming and debugging these specialty processors.

Beginning with the Intel 486 and subsequent Pentium class processors being used in the popular desktop PC platform, the processing power has increased to the point where real time signal processing can now be done using floating point arithmetic and a PC soundcard for analog I/O. The amateur radio community has benefited from these advances with PC Soundcard based applications for SSTV, RTTY, and more recently, PSK31.

My interest in all this was in learning how to develop and program various DSP communications algorithms using a standard Windows[1] based PC platform. It is from these experiments that WinPSK evolved from basically a DSP test bed to a simple functioning program for PSK31. This paper describes in some detail the basic design decisions that were made during this learning process that led to the final program. It is not meant as a definitive reference on PSK31 implementation but just an engineering notebook describing this program. Perhaps others can build on some of the information here to improve this program as well as be motivated to experiment with new modes.

The basic goal was to write a working PSK31 interface program from scratch. Unlike some other HF modes, Peter Martinez has made available very complete specifications for this mode[2],[3]. Also his Windows program "PSKsbw" provided an excellent reference program for verifying and testing various algorithms.

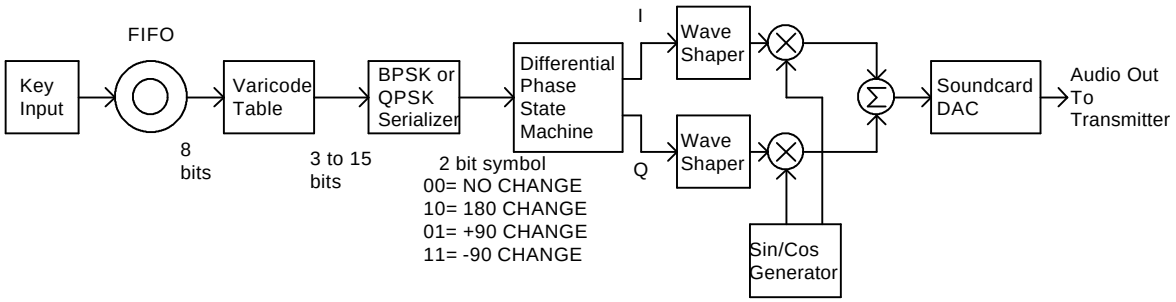
"WinPSK User Guide" is a separate document that describes the user operation of the program. This document only describes the inner workings of the program.

The program will be described in sections starting with signal generation then followed by the reception algorithms. Finally, the overall software architecture and miscellaneous issues will be discussed.

2. WinPSK Signal Generation

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

2.1. Block Diagram



2.2. Input Characters

PSK31 sends and receives 8 bit characters. 0 through 127 are the standard ASCII characters and 128 to 255 are extended characters. WinPSK ignores most control codes below the SP(/0x30) character to reduce some of the garbage from making it to the screen. Some of the Windows controls also behave badly when a control character is sent to them.

2.3. 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.

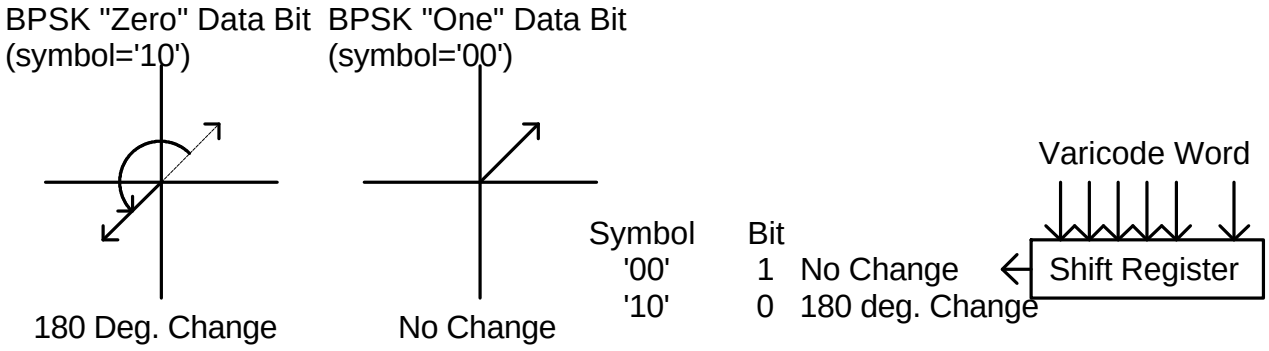
Example bit stream of varicoded character sequence "abc":

…00<u>10110</u>0<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

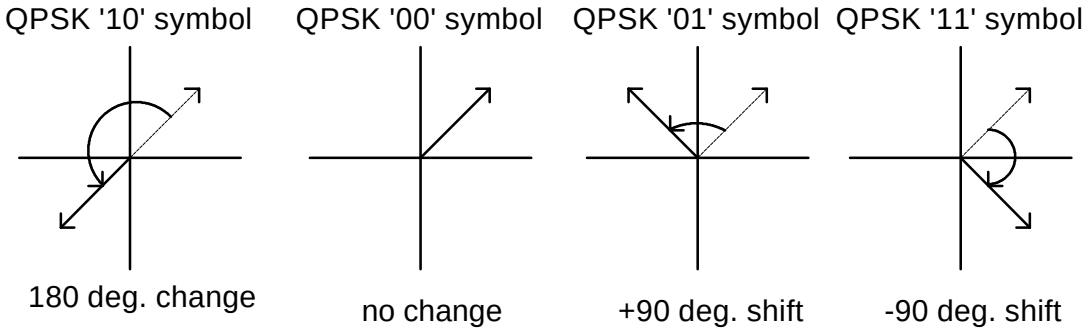
2.4. BPSK Serialization

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.



2.5. 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.



2.5.1. ECC Encoding Method

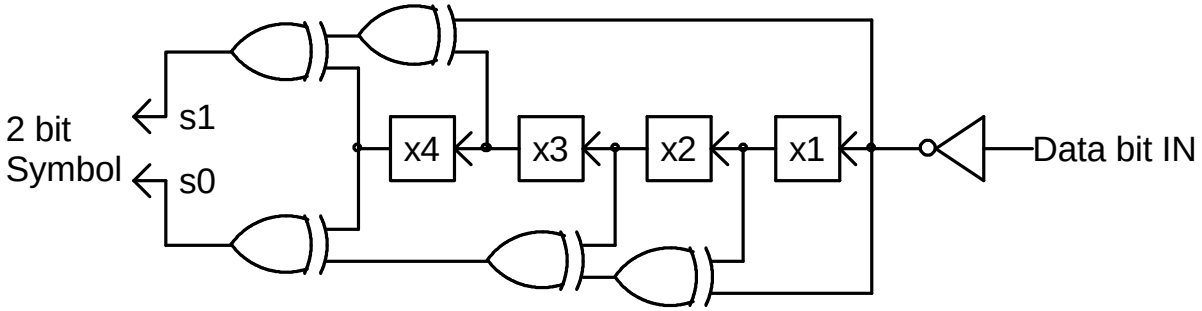
The error correcting coding method used in PSK31 uses convolution codes to essentially "spread out" the redundant information over time. If one were to simply send each bit twice it is easily seen that if an error occurs in one of the bits, there is no way to tell which bit is the correct one so the 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[10] 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:

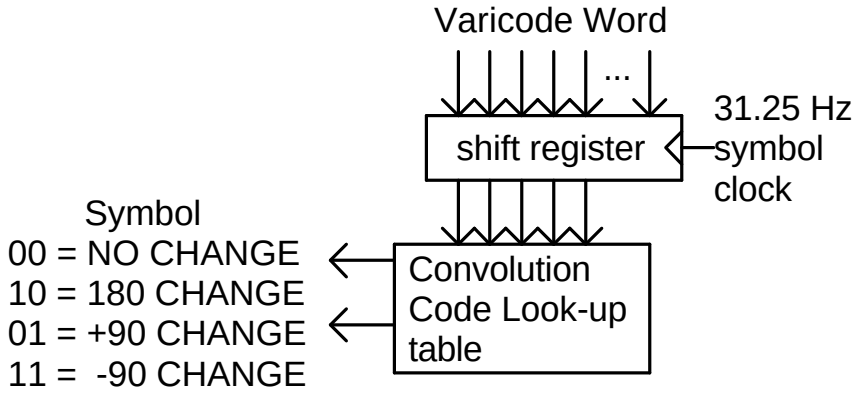
$$G1(x) = x^4 + x^3 + x^0$$

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

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.



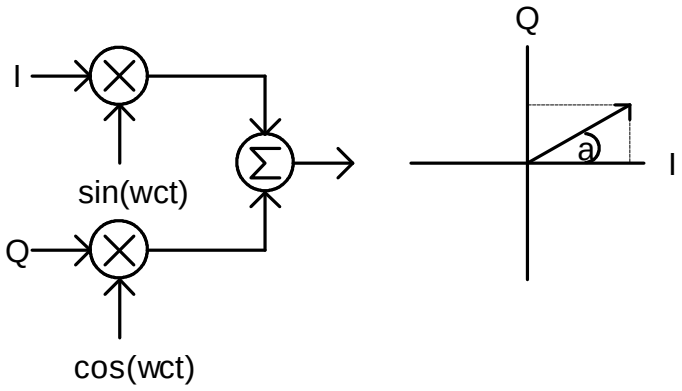
The QPSK encoder is actually implemented using a look-up table rather than using exclusive OR gates.

2.6. 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 WinPSK this is done using state tables.

2.7. 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[4] simulation shows how a BPSK signal could be created using the I/Q method.

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

Symbol frequency $Fs := 31.25$

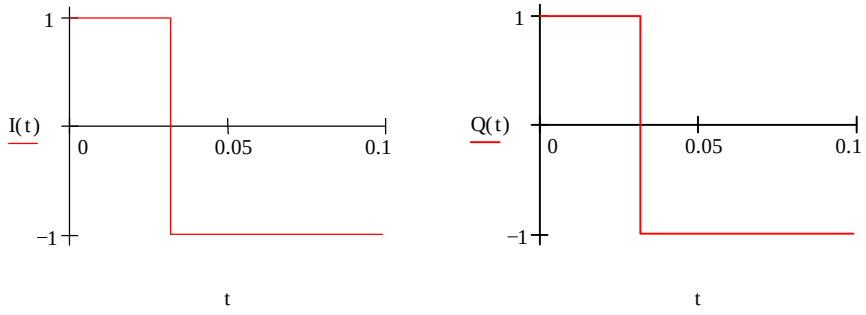
carrier frequency $Fc := 150$

Carrier equations

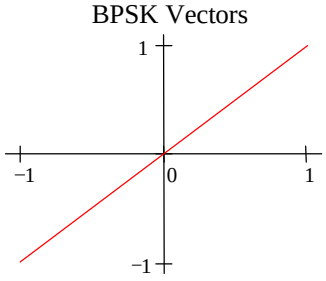
$Ic(t) := A \cdot \sin(2 \cdot \pi \cdot t \cdot Fc)$ $Qc(t) := A \cdot \cos(2 \cdot \pi \cdot t \cdot Fc)$

Modulation equations

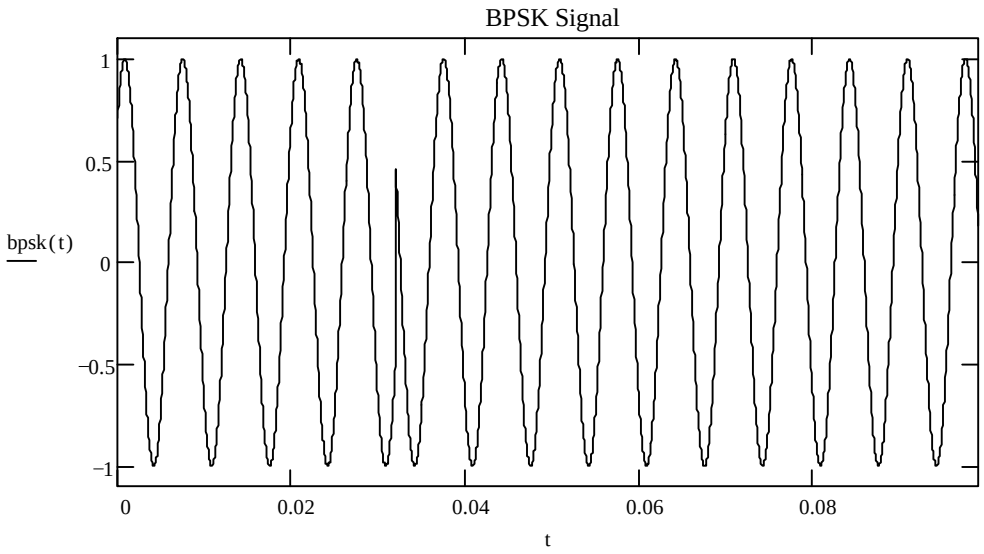
$I(t) := if((t<.032), 1, -1)$ $Q(t) := if((t<.032), 1, -1)$



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

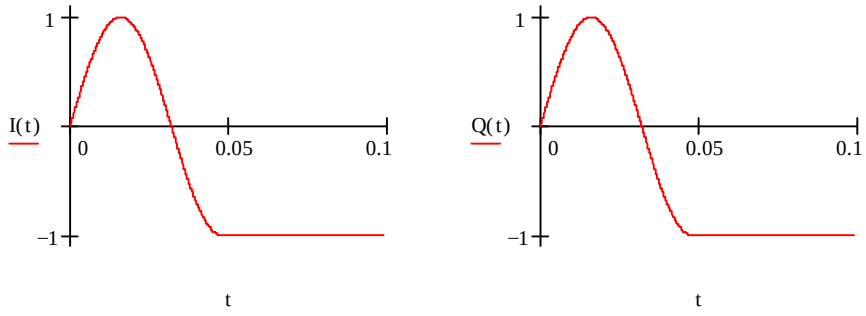
$Fs := 31.25$ Symbol frequency $\qquad t := 0, .000032 .. 0.099$ Plot range

carrier equations

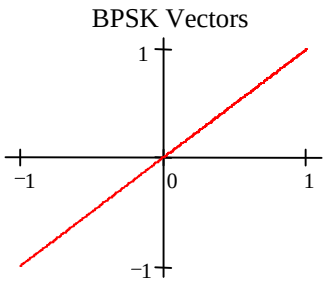
$Ic(t) := A\cdot\sin(2\cdot\pi\cdot t\cdot Fc) \qquad Qc(t) := A\cdot\cos(2\cdot\pi\cdot t\cdot Fc)$

modulation equations

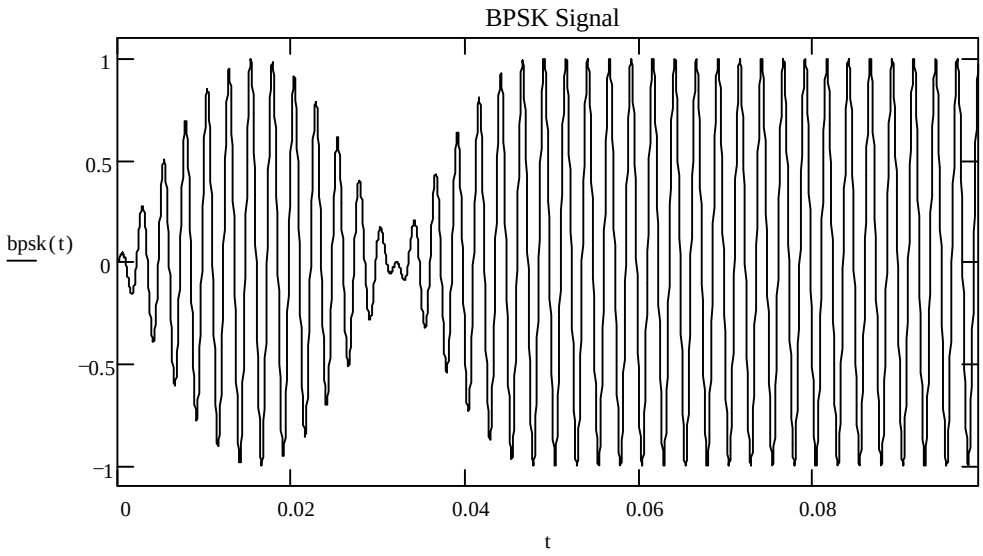
$I(t) := if((t<.048), \sin(\pi\cdot Fs\cdot t), -1) \qquad Q(t) := if((t<.048), \sin(\pi\cdot Fs\cdot t), -1)$



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



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

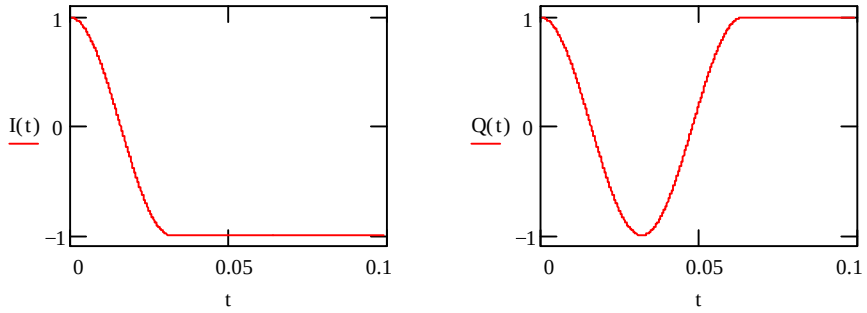


Note the gradual transition from one phase state to the next. This results in a much narrower bandwidth signal without the need for any post filtering. Also it can be seen that the amplitude of the signal is not constant. This means that the transmitter must not compress or limit the audio waveform otherwise 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.

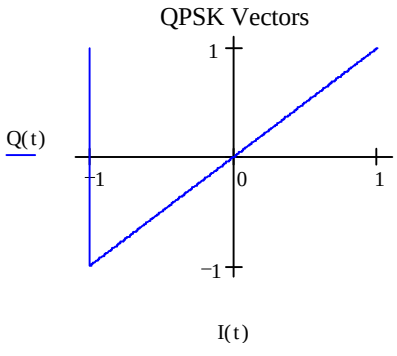
Finally here is a MathCad simulation showing a QPSK signal that changes 180 degrees then by -90 degrees.

Modulation equations

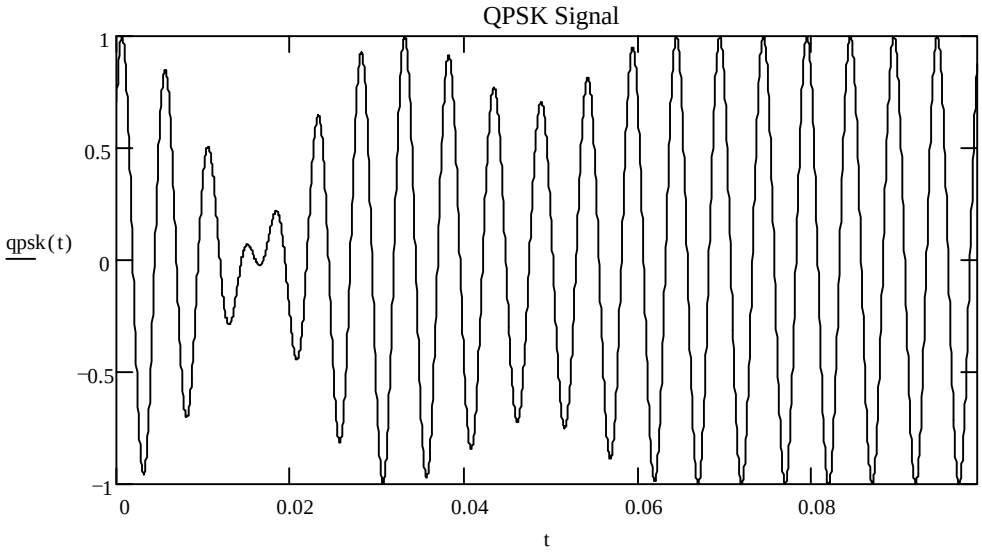
$$I(t) := \text{if}\left(t<.032, \cos\left(\pi\cdot\frac{t}{T}\right), -1\right) \qquad Q(t) := \text{if}\left(t<.064, \cos\left(\pi\cdot\frac{t}{T}\right), 1\right)$$



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



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



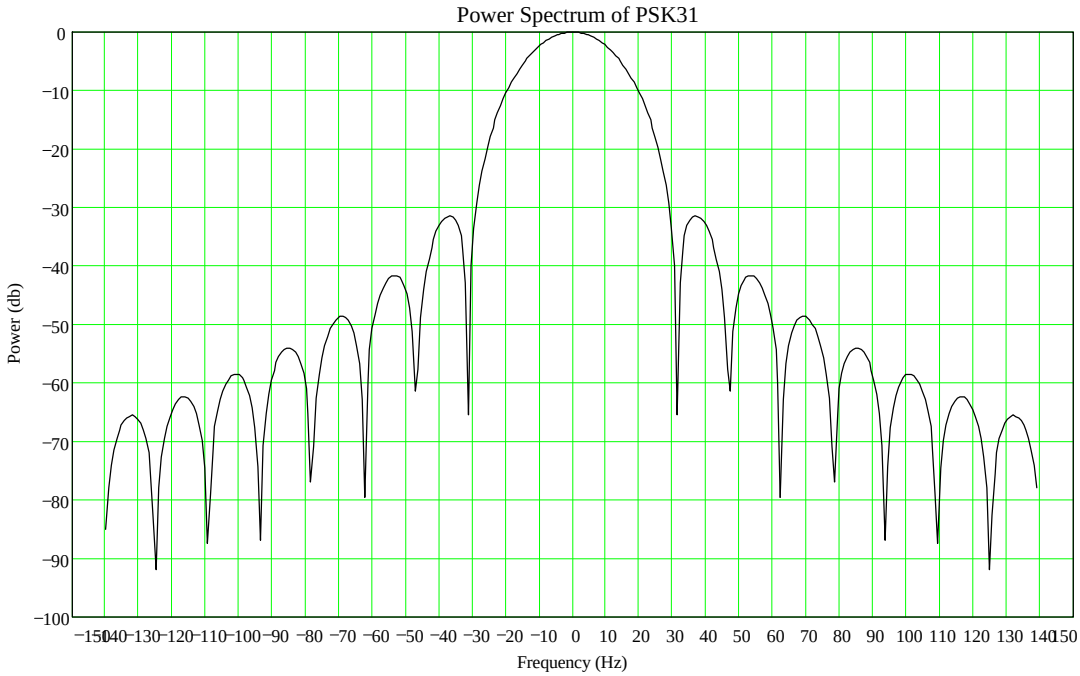
Note that during 90 degree phase changes, the amplitude does not drop all the way to zero as in the 180 degree case.

2.8. Power Spectrum

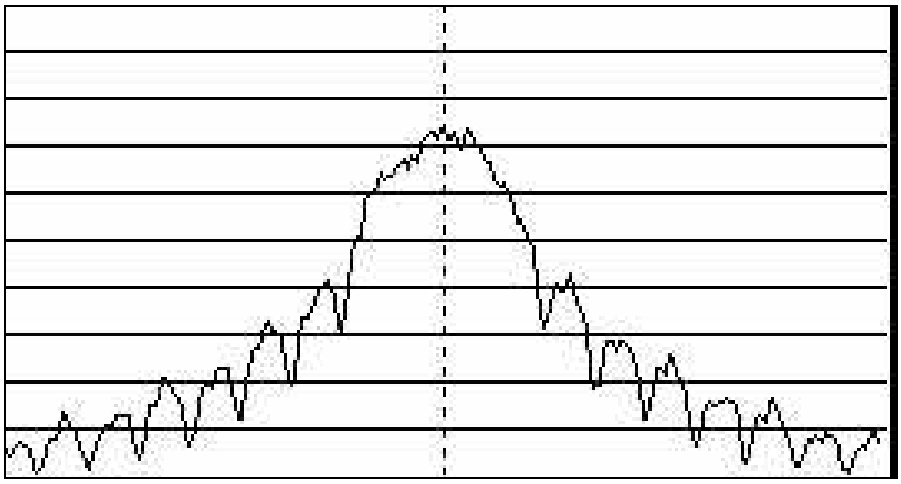
The BPSK/QPSK signal has a power spectrum consisting of a large main lobe centered around the carrier frequency out to a null at the carrier frequency +/- 31.25 Hz. There are multiple lobes extending out to infinity but their amplitudes continue to drop.
Here is a MathCad simulation of the PSK31 power spectrum[6].

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

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



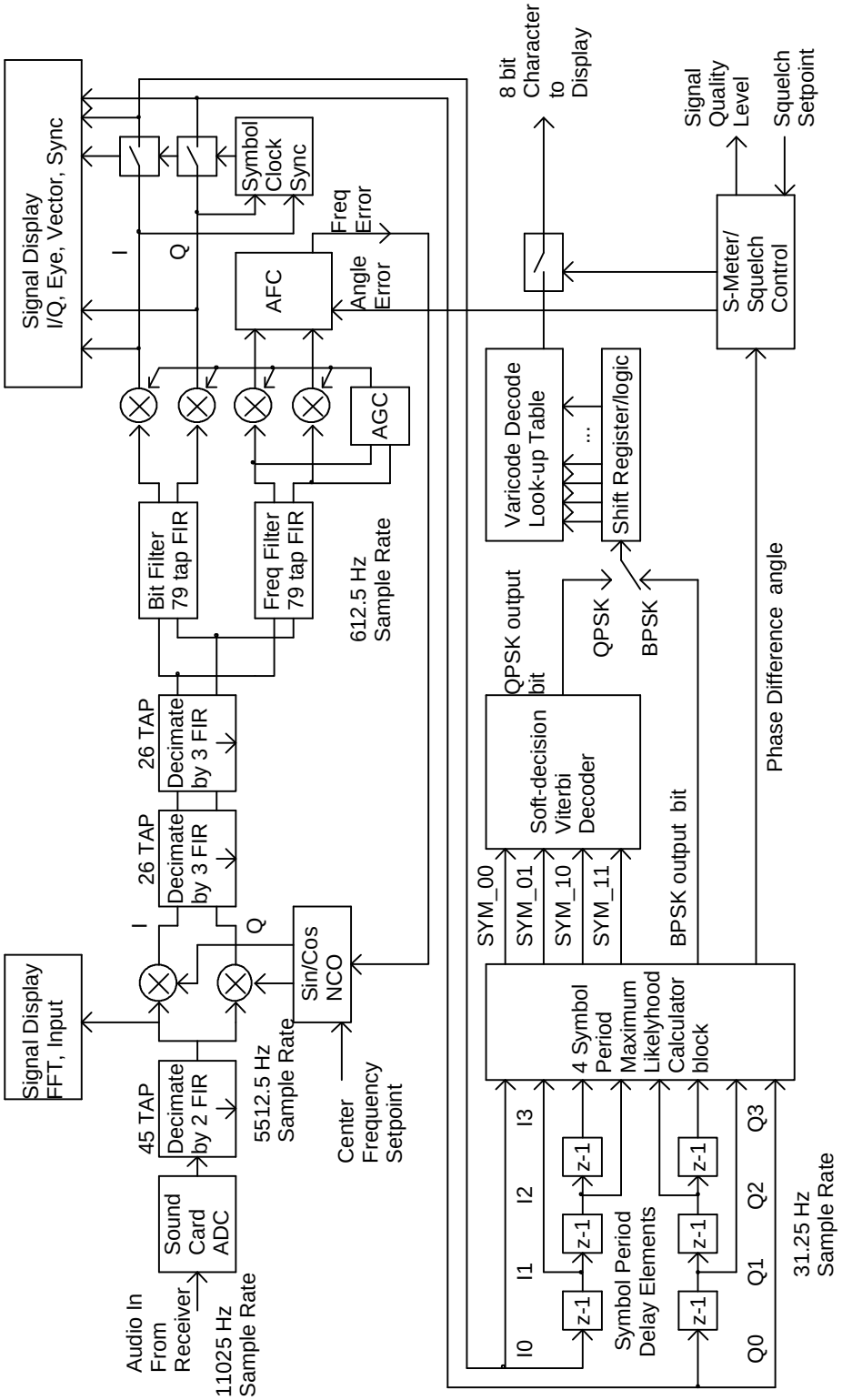
The following FFT scan of a QPSK signal compares favorably with the math model. The vertical divisions are 10 db.



3. WinPSK Signal Detection

3.1. Block Diagram

The following block diagram shows the major functions implemented by WinPSK to receive PSK31 signals. Receiver audio is captured by the PC soundcard and processed into final ASCII characters for display. This schemeis by no means the most optimal method but is the method that evolved from a lot of experimentation with various architectures for WinPSK.

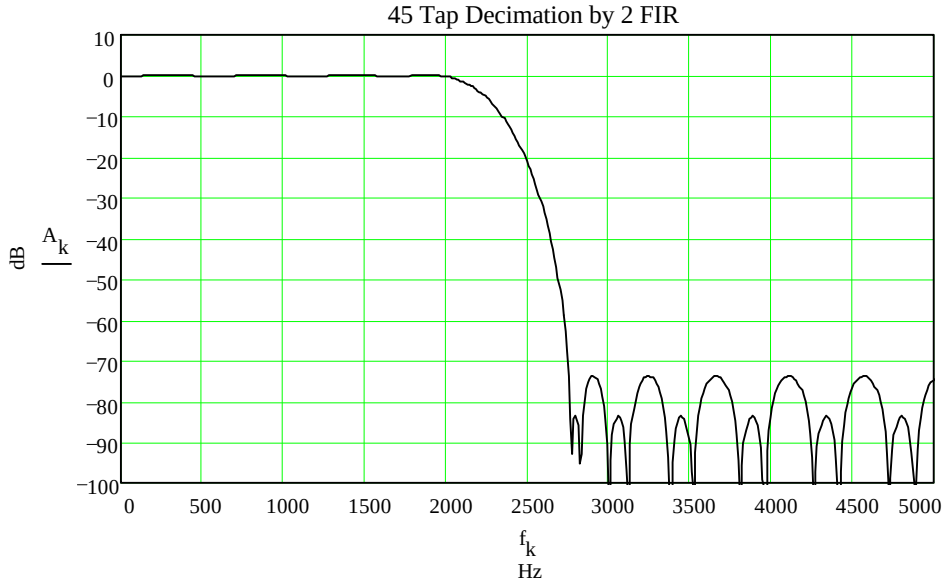


3.2. Soundcard Input

The receiver audio is sampled by the soundcard into 16 bit samples at a 11025 Hz rate and converted into floating point representation for the remainder of the processing.

3.3. Decimation by 2

The first processing block bandlimits the signal to 2700 Hz and performs a sample rate conversion down to 5512.5 Hz. A 45 Tap FIR decimation filter is used to perform this task. The following is a frequency response plot of the filter. The response is flat out to about 2000 Hz which is about the limit the FFT display can accurately display.



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

3.4. 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.

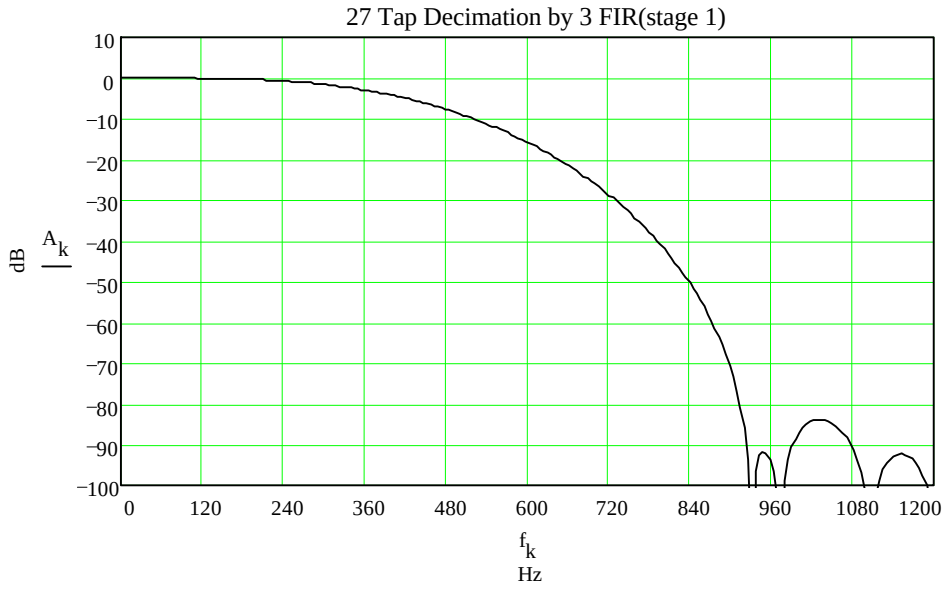
$$I(t) = Input(t)\cos(wt) \qquad Q(t) = Input(t)\sin(wt)$$

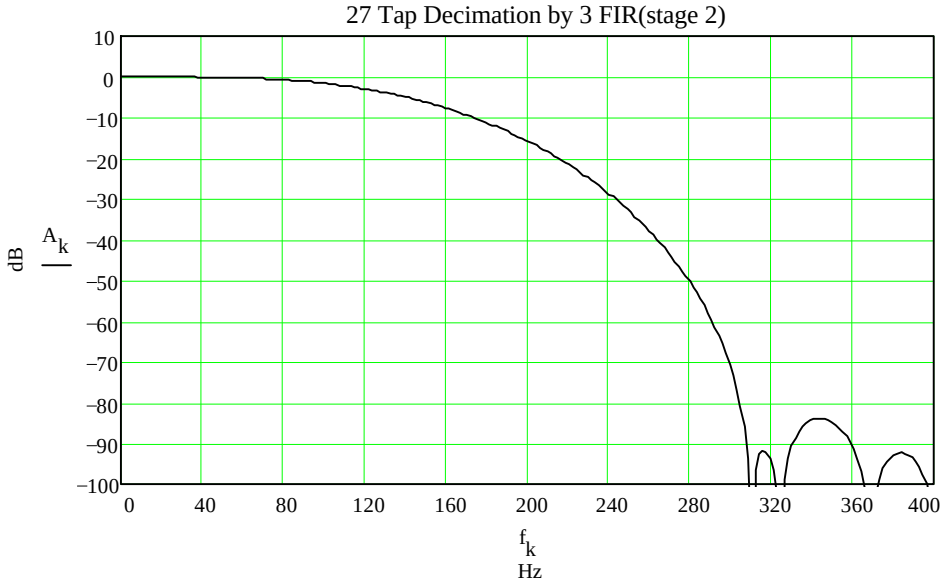
The following code segment performs all these functions:

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

3.5. Decimation by 9

The complex signal is then down sampled again by 9 to further reduce the sampling rate to about 20 times the signal bandwidth. Two stages of decimation by 3 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 3. Since the signal is complex, the filters are run on both the I and Q signal.

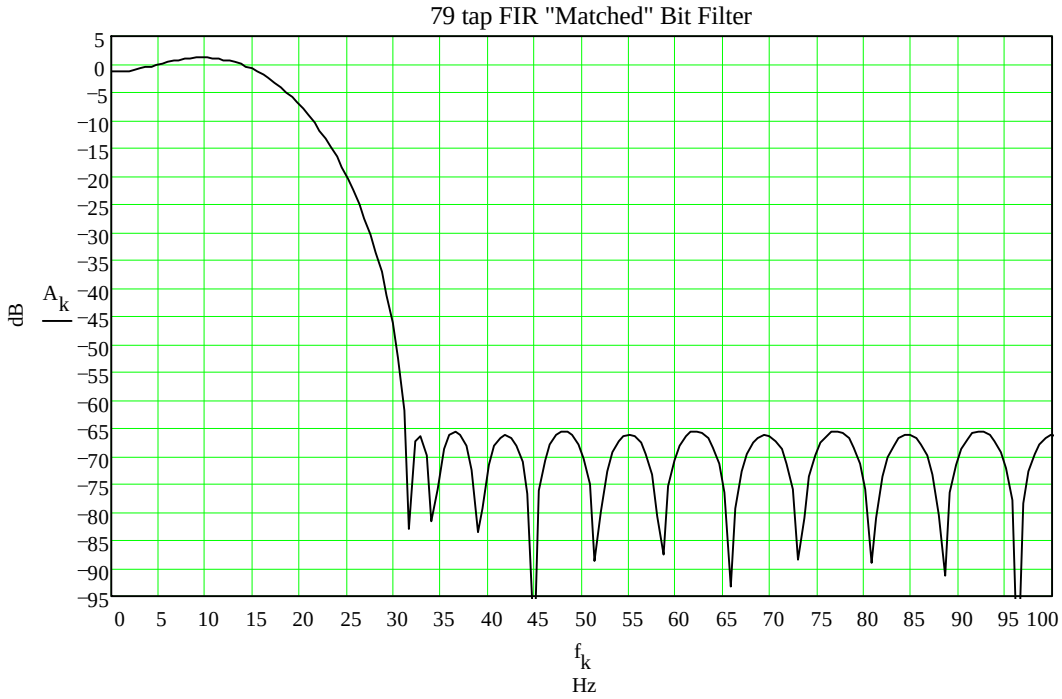




The final sampling frequency is now 11025/18 or 612.5 Hz.

3.6. Matched Data Bit filter

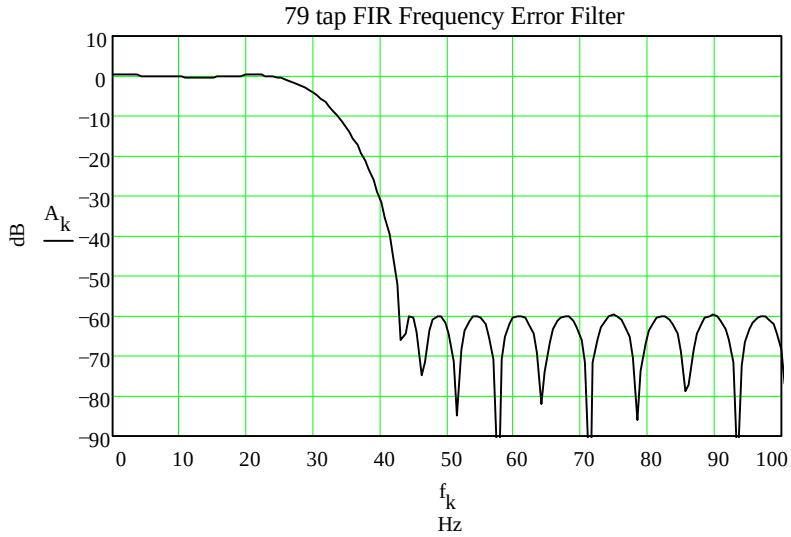
The final system bandwidth is set by this FIR filter. This filter has two purposes. One is to provide a magnitude response that provides the best signal to noise ratio in order to extract the data signal from the noise. The second thing it must do is minimize any ISI(Inter-Symbol Interference) that is 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. I could find no books or papers on the ideal filter for this mode of PSK. By experimentation it appears the ideal filter would a "brickwall" low pass filter with a flat magnitude response out to one-half the bit rate of 31.25 Hz or 15.625 Hz. It must also have a minimum amount of delay. These requirements seem to be almost mutually exclusive since the more ideal the filter, the longer the delay. 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.

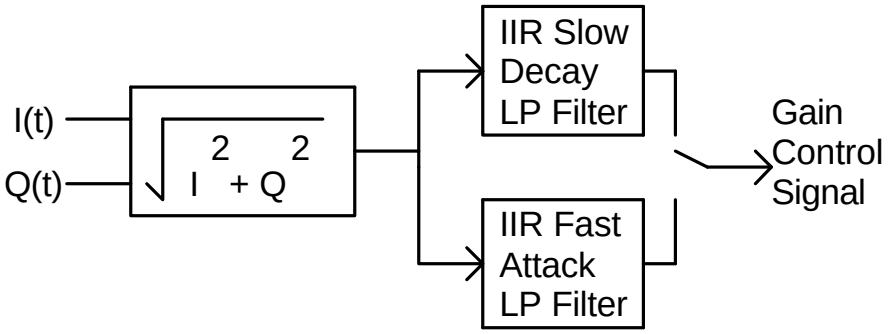
3.7. Frequency Error filter

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.



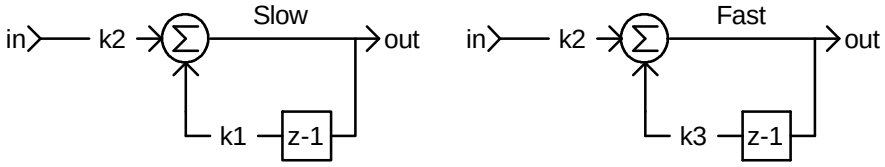
3.8. AGC

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 it's poor frequency response as can be seen in the following plots.

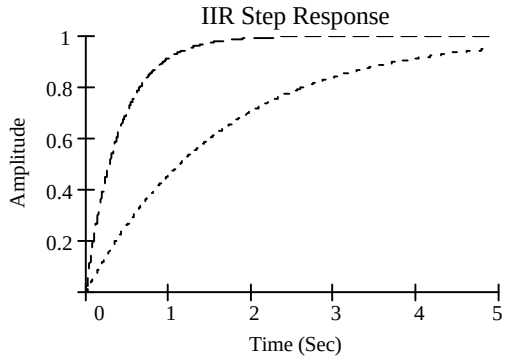


$KS := 1000$ $KF := 250$

$$t(n) := n \cdot \frac{18}{11025}$$

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

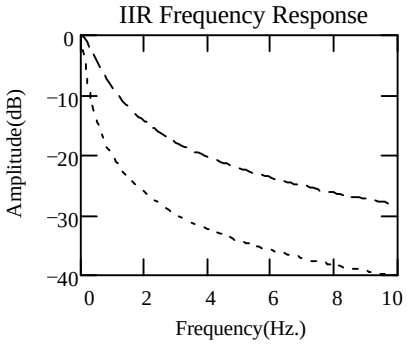
$ys_n := k1 \cdot ys_{n-1} + k2 \cdot x_n$ $yf_n := k3 \cdot yf_{n-1} + k4 \cdot x_n$



$Fs := \frac{11025}{18}$ $T := \frac{1}{Fs}$ $fs := 10$ $\delta f := 0.01 \cdot fs$

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

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



The following code segment performs the AGC function:

```
mag = sqrt(Samp.x*Samp.x + Samp.y*Samp.y);
if( mag > m_AGCave )
      m_AGCave = (1.0-1.0/250.0)*m_AGCave + (1.0/250.0)*mag;
else
      m_AGCave = (1.0-1.0/1000.0)*m_AGCave + (1.0/1000.0)*mag;
if( m_AGCave >= 1.0 )   // divide signal by ave if not almost zero
{
      m_BitSignal.x /= m_AGCave;
      m_BitSignal.y /= m_AGCave;
      m_FreqSignal.x /= m_AGCave;
      m_FreqSignal.y /= m_AGCave;
}
```

3.9. Frequency Error Detection/Correction

The first attempt at frequency control used a modified Costas loop PLL in order to achieve coherent signal detection[5]. The method worked for BPSK but required a very narrow bandwidth loop filter in order to lock on QPSK signals. This work was abandoned in favor of wideband frequency locking and non-coherent PSK detection. Any benefits that coherent detection would have gained would probably be lost in the HF environment.

The AFC is now 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.

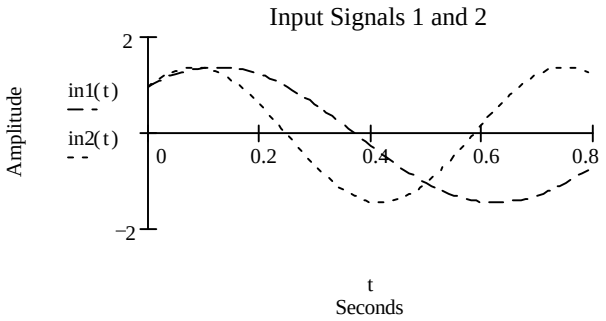
The following Mathcad simulation shows this relationship using two different frequency signals.

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

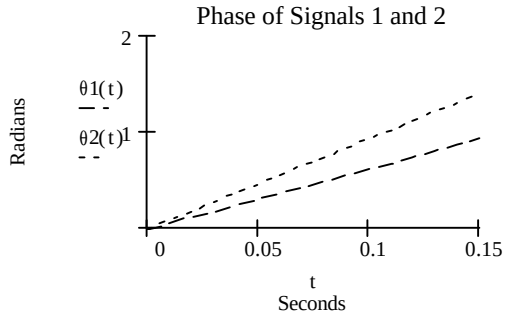
$I1(t) := \cos(2 \cdot \pi \cdot F1 \cdot t)$ $\qquad$ $I2(t) := \cos(2 \cdot \pi \cdot F2 \cdot t)$

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

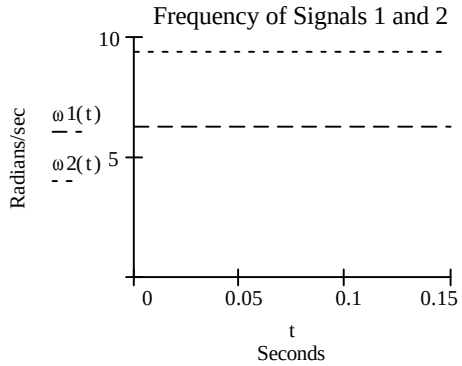
$in1(t) := I1(t) + Q1(t)$ $\qquad$ $in2(t) := I2(t) + Q2(t)$



$$\theta 1(t) := \operatorname{atan}\left(\frac{Q1(t)}{I1(t)}\right) \qquad \theta 2(t) := \operatorname{atan}\left(\frac{Q2(t)}{I2(t)}\right)$$



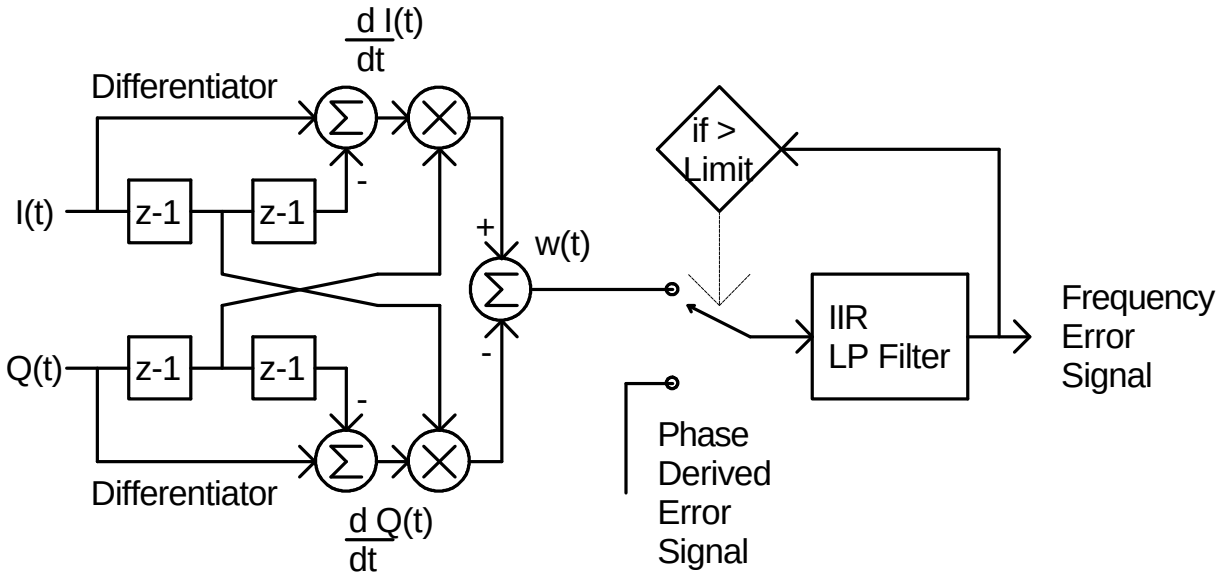
$$\omega 1(t) := \frac{d}{dt}\theta 1(t) \qquad \omega 2(t) := \frac{d}{dt}\theta 2(t)$$



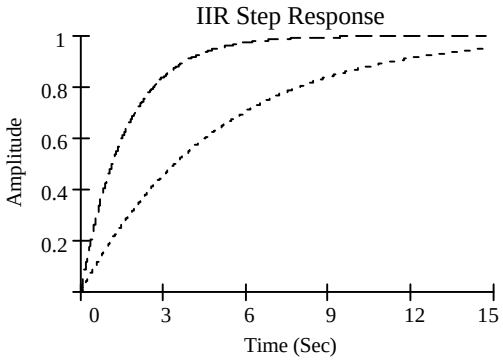
From a dusty calculus book[6] the following identity was obtained that gives the signal frequency as a function of the I and Q signals and their derivatives without the use of the atan() function.

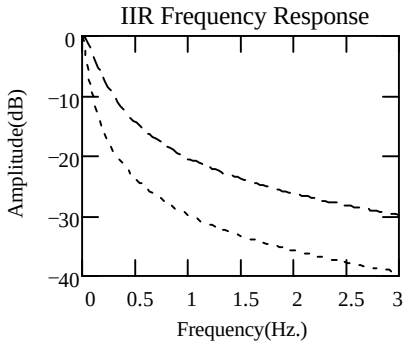
$$\frac{d}{dt}\operatorname{atan}\left(\frac{Q(t)}{I(t)}\right) = \frac{1}{1+\left(\frac{Q(t)}{I(t)}\right)^2}\cdot\frac{d}{dt}\frac{Q(t)}{I(t)} = \frac{I(t)\cdot\frac{d}{dt}Q(t) - Q(t)\cdot\frac{d}{dt}I(t)}{I(t)^2 + Q(t)^2}$$

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:

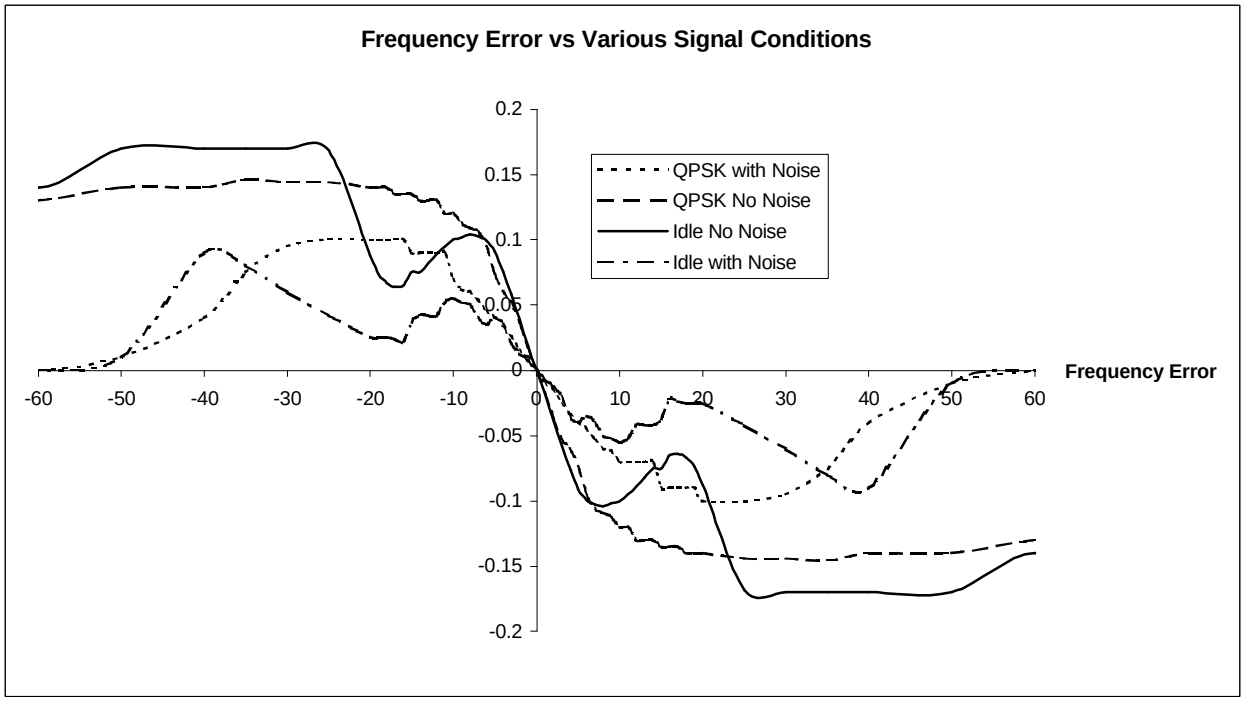


The IIR LP filter are the same type as used in the AGC section with different time constants.





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.



The differential frequency error was originally used to correct the master NCO by itself. Now a secondary phase derived error signal is used once the error gets within 3 Hz. The differential frequency error signal is used when the error signal is large, then the phase derived frequency error kicks in when the error becomes small.

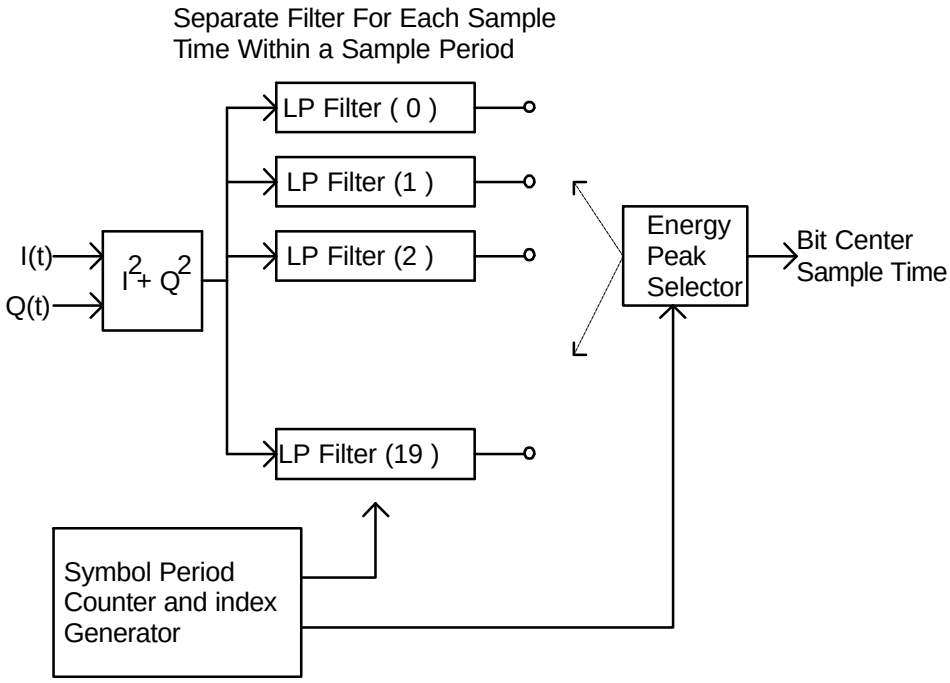
This secondary error signal is derived from the difference angle of the baseband signal (described later). This error signal is generated by how far from the ideal 0 or π phase position the signal is. If the frequency is off, then the phase difference of the PSK31 signal will be rotated from the ideal 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 used once the main frequency error has dropped within 3 Hz.

The following is a code snippet from the AFC control:

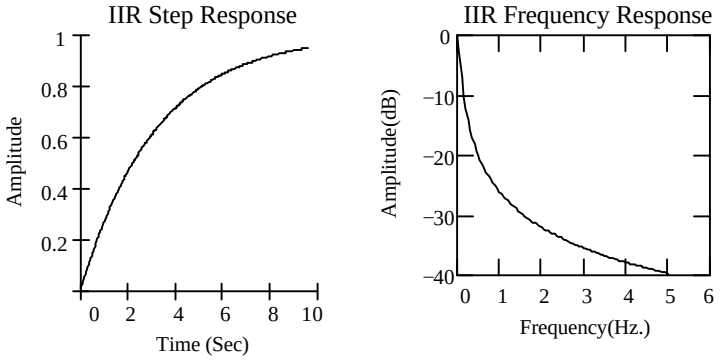
```
ferror = (IQ.x - m_z2.x) * m_z1.y - (IQ.y - m_z2.y) * m_z1.x;
m_z2.x = m_z1.x;
m_z2.y = m_z1.y;
m_z1.x = IQ.x;
m_z1.y = IQ.y;
if( ferror > .15 )            //clamp range
       ferror = .15;
if( ferror < -.15 )
       ferror = -.15;
// error at this point is abt .016 per Hz error
#define FERRLPK (3000.0)      //freq erro LP filter time constant
#define FERRLPG (4)           // freq filter Gain constant(1/4)
#define FCROSS (.048)         //freq where changes to using phase derived freq error
#define FEMIN (FCROSS/FERRLPG)
if( (m_FerAve < -FEMIN) || (m_FerAve > FEMIN) )      //if error >FEMIN then use ferror
       m_FerAve = (1.0-1.0/FERRLPK)*m_FerAve + (1.0/(FERRLPG*FERRLPK))*ferror;
else                                                 //else use phase derived error
{
       if( (m_QFreqError > -FCROSS) && (m_FerAve <= 0.0) )
              m_FerAve = (1.0-1.0/FERRLPK)*m_FerAve + (1.0/(FERRLPG*FERRLPK))*m_QFreqError;
       else
              m_FerAve = (1.0-1.0/FERRLPK)*m_FerAve + (1.0/(FERRLPG*FERRLPK))*ferror;
       if( (m_QFreqError < FCROSS) && (m_FerAve >= 0.0) )
              m_FerAve = (1.0-1.0/FERRLPK)*m_FerAve + (1.0/(FERRLPG*FERRLPK))*m_QFreqError;
       else
              m_FerAve = (1.0-1.0/FERRLPK)*m_FerAve + (1.0/(FERRLPG*FERRLPK))*ferror;
}
```

3.10. 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[7]. 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 about 20 samples per symbol at the 612.5 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:

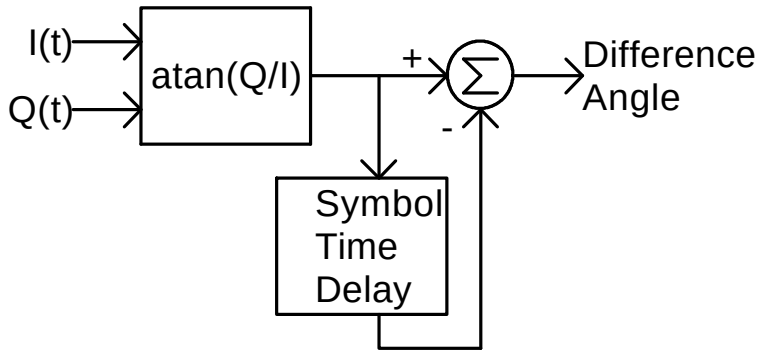


The following is a segment from the bit sync routine:

```
energy = (sample.x*sample.x) + (sample.y*sample.y);
m_SyncAve[BitPos] = (1.0-1.0/100.0)*m_SyncAve[BitPos] + (1.0/100.0)*energy;
if( BitPos == m_PkPos )       // see if at middle of symbol
{
        Trigger = TRUE;
        pDoc->m_SyncHist[m_PkPos] = (INT)(10000.0*m_SyncAve[m_PkPos]);
}
else
{
        Trigger = FALSE;
        pDoc->m_SyncHist[BitPos] = (INT)(7000.0*m_SyncAve[BitPos]);
}
if( BitPos == HALF_TBL[m_PkPos] )     //don't change pk pos until
        m_PkPos = m_NewPkPos;         // halfway into next bit.
BitPos++;
m_BitPhasePos += (m_BitPhaseInc);
if( m_BitPhasePos >= Ts )
{                                     // here every symbol time
        m_BitPhasePos -= Ts;          //keep phase bounded
        BitPos = 0;
        max = -1e10;
        for( INT i=0; i<19; i++)      //find maximum energy pk
        {
                energy = m_SyncAve[i];
                if( energy > max )
                {
                        m_NewPkPos = i;
                        max = energy;
                }
        }
        m_LastPkPos = m_PkPos;
}
m_BitPos = BitPos;
```

3.11. Squelch Function

To implement a squelch function, a measure of signal quality is needed. It was hoped that by using some statistical measures on the symbol sample bin energies, that a squelch function could be derived. Unfortunately, the distribution on these bins does not change very much until the signal gets very noisy. What is needed is a more linear measure of signal quality. The answer came by looking at the vector display of Peter's PSKsbw program. 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.

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

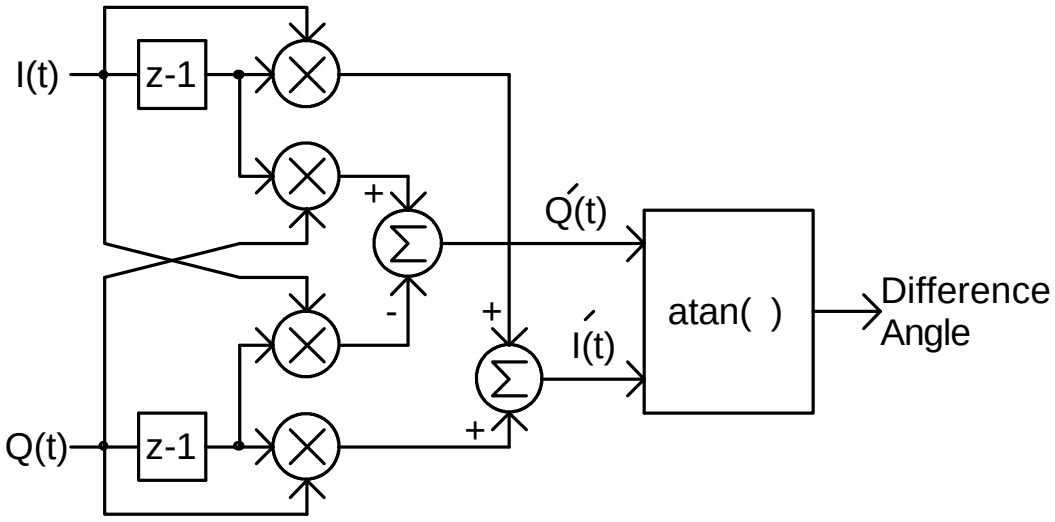
$$Z_k = Y_k \cdot \left(\overline{Y}\right)_{k-1} = A_k \cdot A_{k-1} \cdot e^{j\left(\theta_k - \theta_{k-1}\right)}$$

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

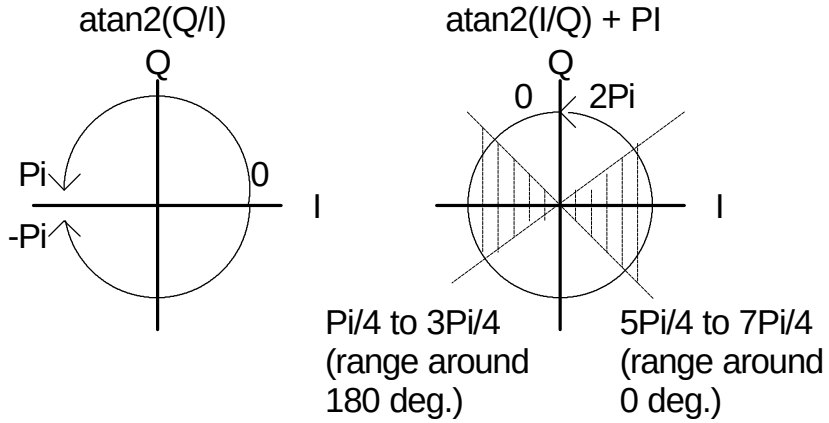
In terms of the complex coordinates I and Q,

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

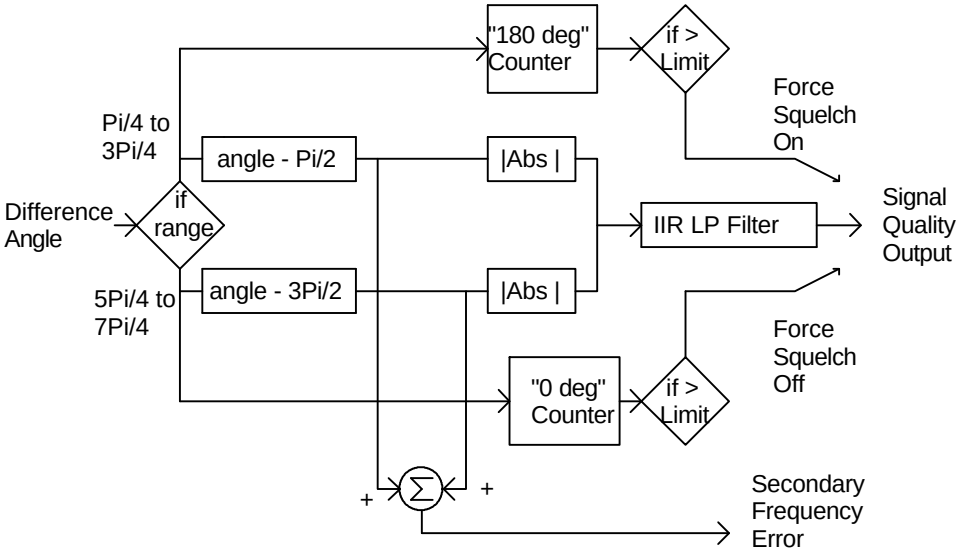
In diagram form this implementation is show below.



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.



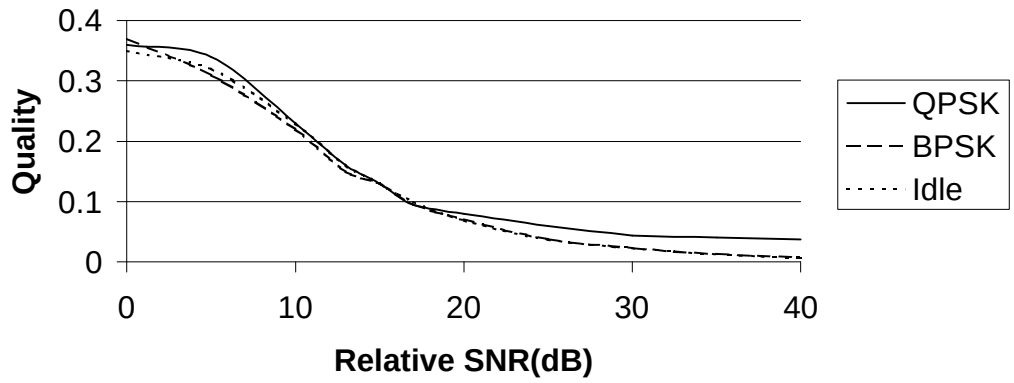
The basic signal quality scheme is then implemented as follows:



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.

In WinPSK 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.

The following is the basic squelch/signal quality function:

```
if( (angle >= PHZ_180_MIN) && (angle <= PHZ_180_MAX)  )
{       //look +/-45 deg. around 180 deg.
        if((pDoc->m_pSettings->m_ModType == QPSK_MODE) && (pDoc->m_pSettings->m_UseLSB))
                temp = PI2/4.0 - angle;
        else
                temp = angle - PI2/4.0;
        m_QFreqError = (1.0/12.5)*temp;              //normalize same as freq error
        temp = fabs(temp);
        if( temp < m_DevAve)
                m_DevAve=  (1.0-1.0/50.0)*m_DevAve + (1.0/(50.0))*temp;
        else
                m_DevAve=  (1.0-1.0/100.0)*m_DevAve + (1.0/(100.0))*temp;
        m_OffCount = 0;
        if(m_OnCount > 20 )                          // fast squelch counter
                m_DevAve = 0.1;
        else
                m_OnCount++;
}
else
        if( (angle >= PHZ_0_MIN) && (angle <= PHZ_0_MAX) )
        {               //look +/-45 deg around 0 deg.
                if((pDoc->m_pSettings->m_ModType == QPSK_MODE) && (pDoc->m_pSettings->m_UseLSB))
                        temp = 3*PI2/4.0 - angle;
                else
                        temp = angle - 3*PI2/4.0;
                m_QFreqError = (1.0/12.5)*temp;              //normalize same as freq error
                temp = fabs(temp);
                if( temp < m_DevAve)
                        m_DevAve=  (1.0-1.0/50.0)*m_DevAve + (1.0/(50.0))*temp;
                else
                        m_DevAve=  (1.0-1.0/100.0)*m_DevAve + (1.0/(100.0))*temp;
                        m_OnCount = 0;
                        if(m_OffCount > 20 )
                                m_DevAve = 0.4;
                        else
                                m_OffCount++;
        }
pDoc->m_SquelchLevel = 100 - (INT)(250.0*m_DevAve);
if(pDoc->m_SquelchLevel > pDoc->m_pSettings->m_SQThreshold )
        pDoc->m_SquelchOn = TRUE;
else
        pDoc->m_SquelchOn = FALSE;
```

3.12. Symbol Decoding

The next step is to convert the I and Q signals back into the four possible symbols (two for BPSK). One could use the difference angle as described in the squelch section and find the nearest 0, 90, -90, or 180 degree position and that would be the symbol to use in the decoder. This is actually implemented in WinPSK and used if the user de-selects the following experimental symbol decoder.

An interesting book[8] was written by Yuri Okunev that describes an algorithmic method of decoding symbols using data over several symbol times that is claimed to have nearly the same performance as the ideal coherent processing methods.
The following description will cause the math/DSP experts to cringe so one should go to the above reference for a complete explanation.

Let all possible original transmitted signals over k-1 symbol times be:

$S_i(t) = SI_i(t)\cos(j) + SQ_i(t)\sin(j)$ Where i = 0,2,3…k-1 and φ is an initial unknown phase.

The received signal is $x(t) = S_i(t) + n(t)$ where n(t) is White Gaussian Noise.

The error probability will be minimum if the receiver picks the Signal $S_i(t)$ such that the following inequality is correct for any $i \neq j$.

$$\left[\int_0^{(k+1)T} x(t)SI_i(t)dt\right]^2 + \left[\int_0^{(k+1)T} x(t)SQ_i(t)dt\right]^2 > \left[\int_0^{(k+1)T} x(t)SI_j(t)dt\right]^2 + \left[\int_0^{(k+1)T} x(t)SQ_j(t)dt\right]^2$$

Another way to look at it is to find the maximum value for all possibilities of $S_i(t)$ of the left side of the inequality.

$$v_i = \left[\int_0^{(k+1)T} x(t)SI_i(t)dt\right]^2 + \left[\int_0^{(k+1)T} x(t)SQ_i(t)dt\right]^2$$

The integrals can be split into separate integrals for each symbol time spanned by the process which in this case is (k-1)T.

$$v_j = \left[\int_{(n-k-1)T}^{(n-k)T} x(t)SI(t)_j\,dt + \int_{(n-k)T}^{(n-k+1)T} x(t)SI(t)_j\,dt + \ldots\ldots \int_{(n-1)T}^{nT} x(t)SI(t)_j\,dt\right]^2$$
$$+ \left[\int_{(n-k-1)T}^{(n-k)T} x(t)SQ(t)_j\,dt + \int_{(n-k)T}^{(n-k+1)T} x(t)SQ(t)_j\,dt + \ldots\ldots \int_{(n-1)T}^{nT} x(t)SQ(t)_j\,dt\right]^2$$

Now $SI_i(t) = \cos(wt + \Delta q_i)$ the projection of S(t) on the real axis and $SQ_i(t) = \sin(wt + \Delta q_i)$ the projection of S(t) on the imaginary axis where $\Delta\theta$ is the phase information for that symbol period.

$$v_j = \left[\int_{(n-k-1)T}^{(n-k)T} x(t)\cos(wt + \Delta q_{(n-k+1)})dt + \int_{(n-k)T}^{(n-k+1)T} x(t)\cos(wt + \Delta q_{(n-k+2)})dt + \ldots\ldots \int_{(n-1)T}^{nT} x(t)\cos(wt + \Delta q_n)dt \right]^2$$

$$+ \left[\int_{(n-k-1)T}^{(n-k)T} x(t)\sin(wt + \Delta q_{(n-k+1)})dt + \int_{(n-k)T}^{(n-k+1)T} x(t)\sin(wt + \Delta q_{(n-k+2)})dt + \ldots\ldots \int_{(n-1)T}^{nT} x(t)\sin(wt + \Delta q_n)dt \right]^2$$

For BPSK signals, $\Delta\theta$ can be 0 or π.
For QPSK signals, $\Delta\theta$ can be 0, $\pi/2$, $-\pi/2$, or π

3.12.1. **BPSK**

3.12.1.1. Maximum Likelihood Detector

For example let's look at a BPSK signal over 3 symbol times(k=0,1,2). The table shows all possible signal variants.

(n-2) signal phase	(n-1) signal phase	(n) signal phase	$\Delta\theta$ at sample n
ωt	$\omega t+0$	$\omega t+0$	0
ωt	$\omega t+0$	$\omega t+\pi$	π
ωt	$\omega t+\pi$	$\omega t+0$	π
ωt	$\omega t+\pi$	$\omega t+\pi$	0

Using trig identities and substituting all possible $\Delta\theta$'s for a signal over k-1 symbol times, one can represent the above in terms of the I(t) and Q(t) signals.

$$\sin(wt + p) = -\sin(wt) \qquad \cos(wt + p) = -\cos(wt)$$

so substituting all possible angle variants:

$$v_1 = \left[\int_{(n-1)T}^{(n-2)T} x(t)\cos(wt)dt + \int_{(n-2)T}^{(n-1)T} x(t)\cos(wt + 0)dt + \int_{(n-1)T}^{nT} x(t)\cos(wt + 0)dt \right]^2$$

$$+ \left[\int_{(n-1)T}^{(n-2)T} x(t)\sin(wt)dt + \int_{(n-2)T}^{(n-1)T} x(t)\sin(wt + 0)dt + \int_{(n-1)T}^{nT} x(t)\sin(wt + 0)dt \right]^2$$

$$v_2 = \left[\int_{(n-1)T}^{(n-2)T} x(t)\cos(wt)dt + \int_{(n-2)T}^{(n-1)T} x(t)\cos(wt + 0)dt - \int_{(n-1)T}^{nT} x(t)\cos(wt)dt \right]^2$$

$$+ \left[\int_{(n-1)T}^{(n-2)T} x(t)\sin(wt)dt + \int_{(n-2)T}^{(n-1)T} x(t)\sin(wt + 0)dt - \int_{(n-1)T}^{nT} x(t)\sin(wt)dt \right]^2$$

and so on for all 4 variants.

Using the I and Q notation for the integrals one can simplify the equation sets.

$$I_n = \int_{(n-1)T}^{nT} x(t)\cos(wt + \Delta q)dt \qquad Q_n = \int_{(n-1)T}^{nT} x(t)\sin(wt + \Delta q)dt$$

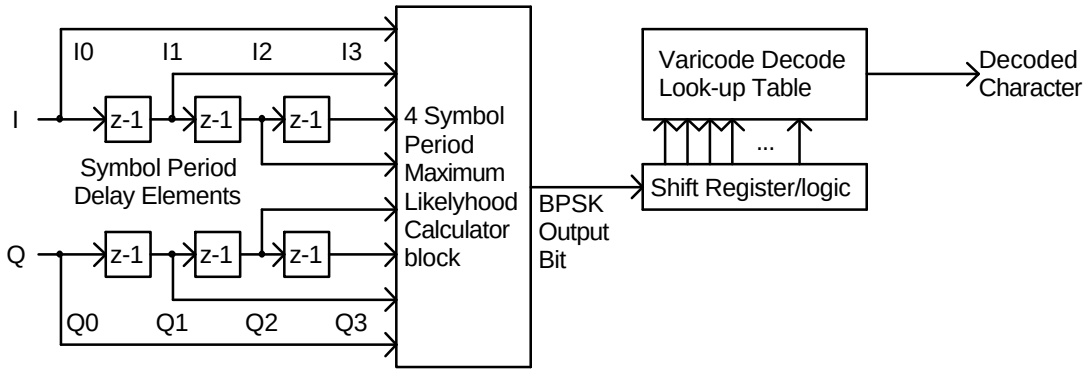
$$v_1 = \left[I_{n-2} + I_{n-1} + I_n\right]^2 + \left[Q_{n-2} + Q_{n-1} + Q_n\right]^2 \quad \text{for } \Delta\theta = 0 \text{ at sample time n}$$

$$v_2 = \left[I_{n-2} + I_{n-1} - I_n\right]^2 + \left[Q_{n-2} + Q_{n-1} - Q_n\right]^2 \quad \text{for } \Delta\theta = \pi \text{ at sample time n}$$

$$v_3 = \left[I_{n-2} - I_{n-1} + I_n\right]^2 + \left[Q_{n-2} - Q_{n-1} + Q_n\right]^2 \quad \text{for } \Delta\theta = \pi \text{ at sample time n}$$

$$v_4 = \left[I_{n-2} - I_{n-1} - I_n\right]^2 + \left[Q_{n-2} - Q_{n-1} - Q_n\right]^2 \quad \text{for } \Delta\theta = 0 \text{ at sample time n}$$

Find the maximum of v1 to v4 and the highest probability $\Delta\theta$ corresponding to that variant is used for the received symbol.
WinPSK looks at all possible variants over 4 symbol periods. This involves looking at a total of 16 variants to make a decision on the BPSK symbol. The new symbol is shifted into a shift register and when an inter-character sequence of two consecutive zeros is received, the complete character is decoded back to ASCII by using a "reverse Varicode" lookup table.



The following is the code to extract the BPSK symbol from all 64 combinations of 4 symbols worth of data:

```
v = pow(m_I3 + m_I2 + m_I1 + m_I0, 2 ) +
       pow(m_Q3 + m_Q2 + m_Q1 + m_Q0, 2 );   // v[1]   0 deg.
max = v; symb = SYM_NOCHANGE;
v = pow(m_I3 - m_I2 + m_I1 + m_I0, 2 ) +
       pow(m_Q3 + m_Q2 + m_Q1 + m_Q0, 2 );   // v[2]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 + m_I2 + m_I1 + m_I0, 2 ) +
       pow(m_Q3 - m_Q2 + m_Q1 + m_Q0, 2 );   // v[3]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 - m_I2 + m_I1 + m_I0, 2 ) +
       pow(m_Q3 - m_Q2 + m_Q1 + m_Q0, 2 );   // v[4]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 + m_I2 - m_I1 - m_I0, 2 ) +
       pow(m_Q3 + m_Q2 - m_Q1 - m_Q0, 2 );   // v[5]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 - m_I2 - m_I1 - m_I0, 2 ) +
       pow(m_Q3 + m_Q2 - m_Q1 - m_Q0, 2 );   // v[6]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 + m_I2 - m_I1 - m_I0, 2 ) +
       pow(m_Q3 - m_Q2 - m_Q1 - m_Q0, 2 );   // v[7]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
v = pow(m_I3 - m_I2 - m_I1 - m_I0, 2 ) +
       pow(m_Q3 - m_Q2 - m_Q1 - m_Q0, 2 );   // v[8]   0 deg.
if( v > max ) { max = v; symb = SYM_NOCHANGE;}
//
v = pow(m_I3 + m_I2 + m_I1 - m_I0, 2 ) +
       pow(m_Q3 + m_Q2 + m_Q1 - m_Q0, 2 );   // v[9]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 - m_I2 + m_I1 - m_I0, 2 ) +
       pow(m_Q3 + m_Q2 + m_Q1 - m_Q0, 2 );   // v[10]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 + m_I2 + m_I1 - m_I0, 2 ) +
       pow(m_Q3 - m_Q2 + m_Q1 - m_Q0, 2 );   // v[11]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 - m_I2 + m_I1 - m_I0, 2 ) +
       pow(m_Q3 - m_Q2 + m_Q1 - m_Q0, 2 );   // v[12]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 + m_I2 - m_I1 + m_I0, 2 ) +
       pow(m_Q3 + m_Q2 - m_Q1 + m_Q0, 2 );   // v[13]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 - m_I2 - m_I1 + m_I0, 2 ) +
       pow(m_Q3 + m_Q2 - m_Q1 + m_Q0, 2 );   // v[14]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 + m_I2 - m_I1 + m_I0, 2 ) +
       pow(m_Q3 - m_Q2 - m_Q1 + m_Q0, 2 );   // v[15]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
v = pow(m_I3 - m_I2 - m_I1 + m_I0, 2 ) +
       pow(m_Q3 - m_Q2 - m_Q1 + m_Q0, 2 );   // v[16]   180 deg.
if( v > max ) { max = v; symb = SYM_P180;}
return !symb;          //bit is inverted symbol
```

3.12.2. QPSK

3.12.2.1. Maximum Likelihood example

For another example let's look at a QPSK signal over 2 symbol times(k=0,1). The table shows all possible signal variants.

(n-1) signal phase	(n) signal phase	$\Delta\theta$ at sample n
ωt	$\omega t+0$	0
ωt	$\omega t+\pi/2$	$+\pi/2$
ωt	$\omega t-\pi/2$	$-\pi/2$
ωt	$\omega t+\pi$	π

One can use the following trig identities to find all variants with respect to I and Q.

$$\sin(wt + p) = -\sin(wt) \qquad \cos(wt + p) = -\cos(wt)$$

$$\sin(wt + \frac{p}{2}) = \cos(wt) \qquad \sin(wt - \frac{p}{2}) = -\cos(wt)$$

$$\cos(wt + \frac{p}{2}) = -\sin(wt) \qquad \cos(wt - \frac{p}{2}) = \sin(wt)$$

$v_1 = \left[I_{n-1} + I_n\right]^2 + \left[Q_{n-1} + Q_n\right]^2$ for $\Delta\theta = 0$ at sample time n

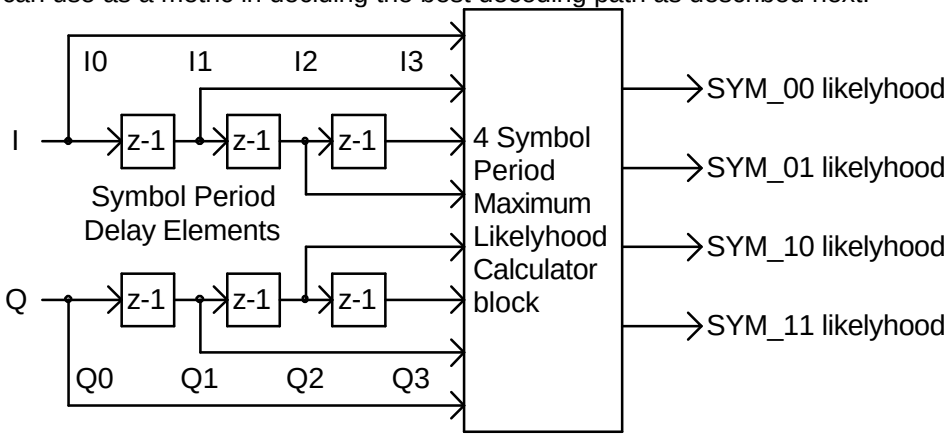
$v_2 = \left[I_{n-1} + Q_n\right]^2 + \left[Q_{n-1} - I_n\right]^2$ for $\Delta\theta = +\pi/2$ at sample time n

$v_3 = \left[I_{n-1} - Q_n\right]^2 + \left[Q_{n-1} + I_n\right]^2$ for $\Delta\theta = -\pi/2$ at sample time n

$v_4 = \left[I_{n-1} - I_n\right]^2 + \left[Q_{n-1} - Q_n\right]^2$ for $\Delta\theta = \pi$ at sample time n

Find the maximum of v1 to v4 and the highest probability $\Delta\theta$ corresponding to that variant is used for the received symbol.

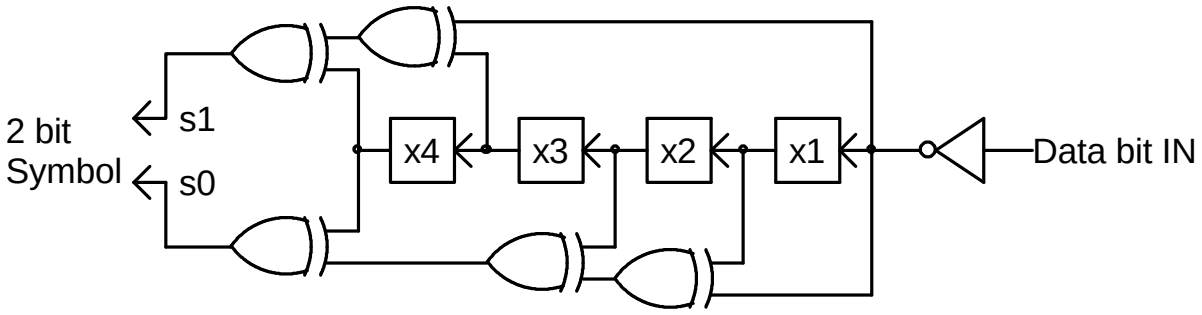
WinPSK looks at all possible variants over 4 symbol periods. This involves looking at a total of 64 variants. Because the next step is to decode the incoming QPSK symbols using a soft Viterbi decoder, the maximum of each symbol possibility(0, +90, -90, and 180 deg.) is calculated, normalized so that the maximum is always equal to one, and -log() is taken to get a probability measure that the Viterbi decoder can use as a metric in deciding the best decoding path as described next.



3.12.2.2. 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[9] so it won't be discussed in much detail here. Reference[10] dedicates an entire chapter on this algorithm and reference[11] provides a good step by step description of the algorithm with lots of trellis drawings.

WinPSK uses a modified implementation of a Viterbi decoder described in an article[12] 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 most likely symbol metric contained in an array(m_QPSKprob[ ]) whose indexed elements contain the value calculated by the symbol decoder. These values are zero for the most likely symbol and a positive value for the other three symbols depending on how confident the symbol decoder was in the decision.
For example:
m_QPSKprob[SYMB_00] = metric for SYMB_00 = 0.3
m_QPSKprob[SYMB_01] = metric for SYMB_01 = 0.0 (SYMB_01 is most likely symbol)
m_QPSKprob[SYMB_10] = metric for SYMB_10 = 0.4 (SYMB_10 is least likely symbol)
m_QPSKprob[SYMB_11] = metric for SYMB_11 = 0.1

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)
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.

```
for(i = 0; i < 32; i++)                 // calculate all possible distances
{                                       //lsb of 'i' is newest bit estimate
        pathdist[i] = m_SurvivorStates[i / 2].Pathdistance + m_QPSKprob[ 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
```

3.13. Display Signals

WinPSK provides several views of the received signal for use in tuning or for determining the quality. All the display signals are calculated separately from the main receiver using a single worker thread to process and plot the data. A single "TAB" control allows switching between several different types of views using the same screen area.

3.13.1. FFT for Spectrum Display

A 2048 point FFT is implemented to obtain frequency information from the received signal. A slightly modified version of Takuya OOURA's original radix 4 FFT software package was used to perform this function. Log output and a Hanning windowing function was added. The data for the FFT is obtained prior to down mixing so the frequency points do not move when changing receive frequency. The FFT data sample rate is 5512..5 Hz so the frequency range is 0 – 2756 Hz with a resolution of about 2.7 Hz. The decimation filter starts to roll off at about 2200 Hz so the display is limited from 100 to 2200 Hz. Display zooming is performed by scaling the index into the FFT's output not by changing FFT size or sample rate.
A standard amplitude vs frequency plot is available as well as a "Waterfall" type in which the amplitude is displayed as intensity/color, the x axis is frequency, and the y axis is time.

3.13.2. Vector Display

A vector display similar to the one used in "PSKsbw" shows the incoming signal phase. It is overlayed on top of the main spectral views. Because WinPSK works on blocks of symbols at a time, the display throws up several vectors at once rather than one at a time. Also the amplitude of the vectors is constant rather than changing with input signal amplitude.
This display is useful for seeing if a signal is BPSK(vertical vectors only) or QPSK(vertical and horizontal vectors). It is also a good test to see if a signal is tuned exactly on frequency. WinPSK's AFC can be fooled and so manual frequency adjustment may be required. Tweaking the frequency for true vertical an horizontal vectors will assure frequency correctness if the AFC has problems on noisy or distorted signals.

3.13.3. Input Signal

The Input signal view is just a scope-like view of the raw input signal prior to down mixing. It's main use is to adjust the soundcard input level to keep from overloading the A/D. The trace color will change from green to red on peaks that get within 3dB of the limit.

3.13.4. Sync histogram

The sync display graphically shows the energy histogram of the bit sync routine. The center of the bit should be at the peak of this display and is shown as an extended line to distinguish it. This display is interesting in that it can be used to show if the received signal bit rate is off frequency (Assuming your own soundcard is on frequency).
If the bit center drifts noticeably, say 15 seconds to drift across the screen, then either yours or the sending soundcard is off frequency by an amount greater than can be explained by simple crystal inaccuracies. From observing several on the air signals it appears that the sample rates on some soundcards is actually 11000 or maybe 11050 Hz instead of the standard 11025. My laptop suffers from this problem and uses 11050 Hz instead of 11025 even though the driver is set to the correct frequency. The WinPSK bit sync algorithm will track this off frequency signal but with a phase lag due to the filtering and so the center of the bit is off causing receiver degradation on noisy signals.
As a countermeasure for this, a clock adjustment offset can be set in WinPSK to compensate for off frequency soundcards. The value is in ppm(parts per million) and a rough estimate of the clock error is calculated while receiving a strong signal and displayed in the program status bar. The problem with this is knowing if it is your soundcard or the other guy's that is off.

4. Windows Program Implementation

4.1. PC/Windows Implementation Issues

The implementation language C/C++ and Microsoft Visual C++ tools were chosen due to existing familiarity and experience. The only real decision to make was whether to program the Windows application using MFC(Microsoft Foundation Classes) or straight API (Application Programming Interface) coding. MFC can provide a lot of functionality with little code but is much harder to customize and you have to deal with the quirky MFC code. Using the API allows the programmer much more control over what is going on but requires a lot of tedious coding to implement even the most basic functionality.

In the end, laziness overcame common sense and the MFC route was undertaken. Looking back I'm still not sure it was the right decision. Perhaps when the hair grows back that I pulled out while dealing with those inane classes, a more objective opinion will ensue.

4.2. Real Time Considerations

The first issue to resolve was how much processing could be done using a PC and Windows. Obviously the basic DSP functions could be done since Peter's and other programs run fine under Windows. One goal was to provide bigger signal displays with more resolution and range. Could this be achieved in addition to all the underlying signal processing that had to occur?

Windows is NOT a real time operating system. The response time to any event such as mouse, key, soundcard, or any other event, is not bounded or specified. It is within "spec" if your mouse hesitates a few hundred milliseconds while your modem disconnects. This at first would seem to be a death sentence for a signal processing application that must process thousands of samples per second without missing a beat. The key to resolving this issue is buffering, lots and lots of buffering. As long as the average processing time your program gets is more than the average amount of time required to perform a task, buffering can be used to fill in the times while the processor is away doing other things.

There still is no guarantee that your application will not starve. Screen savers are probably the most ill behaved applications around in this regard. Some CD rom device drivers and floppy disk drivers can also consume way too much processor time. The best you can do is try to adjust your buffering to ride through most of the processor interruptions. A couple of seconds worth of buffering is probably not out of line. A couple hundred milliseconds may work if no other app is running and you don't move the mouse around much or access the disk. Windows 2000 promises to allow more control over the multitasking quantum times but don't hold your breath.

One feature added to Windows starting with Win95, was multitasking. This allows the programmer to split up tasks into independent sections of code as if a separate processor was being used to execute that task.(With a multiprocessor NT system that can actually be the case) One of the biggest conceptual hurdles in moving from embedded or DOS applications to Windows applications is dealing with the fact that your program is essentially called by the operating system based on messages sent to it by the mouse, keyboard, etc. The use of multitasking threads allows one to write code that does not rely on Windows message processing which is slow and unpredictable with respect to response time.

WinPSK uses two worker threads. One is used to read/write soundcard data and perform all the DSP functions to receive and transmit PSK31 signals. The second thread is used to display signal information graphically. This leaves the main process thread of the program to do the normal mouse, keyboard, windows stuff.

4.3. Float vs. Integer Implementation

Another issue to decide is whether floating point math could be used instead of integer math for the DSP algorithms. Floating point math makes the programming part much easier since overflow and underflow issues diminish as well as built in functions to calculate sines and tangents and stuff are available. The question is whether there is enough horsepower in a Pentium class processor to do such things. A simple test program was written to compare the processing time of doing a simple DSP function, a FIR filter. One thousand samples were run through a one thousand tap FIR using 32 bit integer, 16 bit integer, single precision floating point, and double precision floating point math.

```
//    code segment for timing FIR function
#define TESTBUFSIZE 1000
typedef short TEST_TYPE;
//typedef int TEST_TYPE;
//typedef float TEST_TYPE;
//typedef double TEST_TYPE;
TEST_TYPE* pBuf1;
TEST_TYPE* pBuf2;
TEST_TYPE* pBuf3;
INT i,j;
TEST_TYPE acc;
acc = (TEST_TYPE)0;
for(i=0; i<TESTBUFSIZE; i++)
{
      for(j=0; j<TESTBUFSIZE; j++)
      {
            acc = acc + pBuf1[i] * pBuf2[j];
      }
      pBuf3[i] = acc;
}
```

The following table presents the results on several different processors and operating systems.

Data Type	133 Pentium Win95	400 Pentium II Win98	500 Pentium III NT4.0
16 bit integer	114800 uSec	36200 uSec	29000 uSec
32 bit integer	99400 uSec	9100 uSec	7280 uSec
Float	45800 uSec	8850 uSec	7070 uSec
Double float	69500 uSec	17500 uSec	7110 uSec

From this it was found that single precision floating point is actually faster than integer arithmetic. This sort of makes sense because a separate floating point unit runs in parallel with the main CPU. 16 bit integer arithmetic is much slower than 32 bit probably because the native word size in the Pentium is 32 bits. WinPSK uses double precision floating math since there is not that big a hit in performance and most of the library math functions use doubles for arguments.

4.4. PC Soundcard Settings

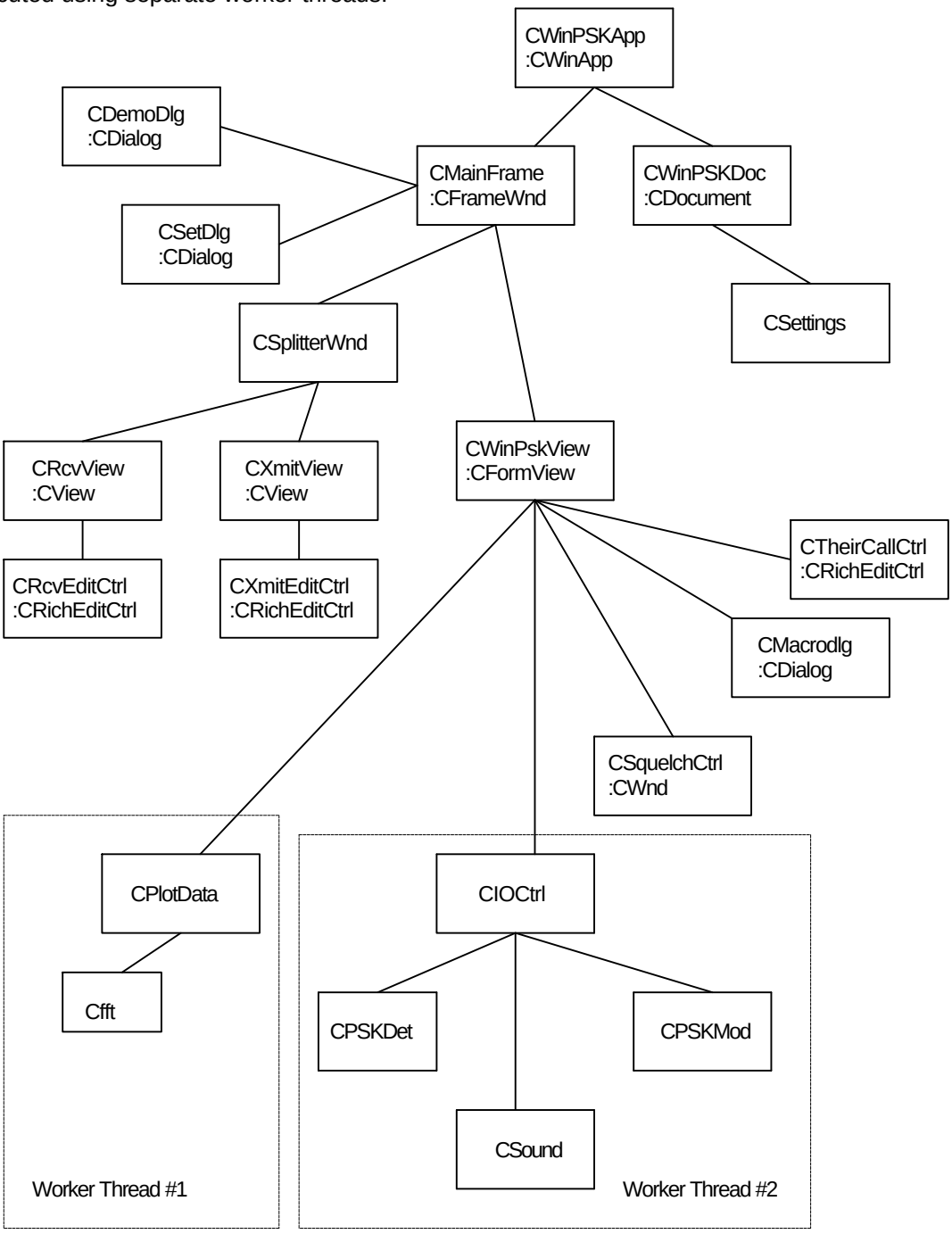
The PC soundcard can be set to various modes of operation. For WinPSK, the fundamental sample rate was chosen to be 11025 Hz. This is a common sample rate that must be supported by any PC soundcard running with Windows. Unfortunately, many soundcards do not implement the 11025 Hz sample rate correctly and round it off to 11000 or 11050 Hz as discussed earlier.

The 16 bit mode is used for all soundcard I/O. This is the number of D/A, A/D resolution bits and should not be confused with the BUS width of the soundcard which could be 8, 16, 32 bits depending on the type. Also the single channel, mono, mode is used rather than stereo.

4.5. Program Structure

4.5.1. Hierarchy Diagram

The following diagram shows the major C++ classes that form the WinPSK program. The base classes are shown for those derived from Microsoft's Foundation Classes. The classes enclosed by dotted lines are executed using separate worker threads.



4.5.2. Class Descriptions

In order to become familiar with Windows programming, one needs to surround yourself with piles of Windows programming books, use the online help reference, and download as many examples of programs as you can. CodeGuru(CodeGuru.com) is one place that has lots of code examples for MFC. Several others exist as well. There are several newsgroups that cater to Windows programming as well. Below is a brief description of the major code blocks. The details of the code are not described here but the code is fairly well documented for someone who is familiar with C++, Windows, and MFC. The code is not textbook stuff so don't think this is the best implementation. This author is still struggling with the basics of all this.

CWinPSKApp

Main Entry point for the program. Initializes the basic window's document-view classes and handles the case of multiple program instances.

CMainFrame

Class provides the outer window frame with system and menu controls. It saves and restores the program screen position and size settings to the registry. Screen setting menus and general program setup dialogs are handled here as well as status bar functionality.

CDemoDlg

This dialog class is used to select a demonstration mode that does not use the soundcard but generates a repeating character string and then decodes it. This useful for playing with the various program features off line. A built in noise generator simulates AWGN with selectable amounts.

CSetDlg

This dialog class is used to enter some general user setup information.

CWinPSKDoc

This class is used primarily to store all sorts of program variables that need to be accessed from other classes and threads. This is one of the few MFC's that are thread safe. Any class that has a Windows message handler cannot be called by any thread except the main process thread.

CSettings

This class is just a structure that holds all the user program settings that need to be saved and restored to disk. There are some class methods that save and restore the class to a file "Settings.dat" that is in the program's path.

CSplitterWnd

This MFC class is used to provide three separate view areas on the program screen. The top view area is for the received text, the middle area is for the transmitted text, and the bottom area holds the program controls, data tuning plots, and macro controls.

CRcvView

A class to create and place the receive text control box in the top splitter window.

CRcvEditCtrl

A CRichEditCtrl derived class that is a read only edit box where all the received text is placed. It handles word wrap, color keying, drag and drop, clipboard operations, etc.

CXmitView

A class to create and place the transmit text control box in the middle splitter window.

CXmitEditCtrl

A CRichEditCtrl derived class that is an edit box where all the text to be transmitted is entered either by typing, drag and drop, clipboard, or macro insertion. It handles word wrap, color keying, drag and drop, clipboard operations, etc. As text is transmitted, it can be color coded to distinguish it from text that has not yet been sent. Back spacing into unsent text deletes the text in the edit box. Back spacing into the sent text string transmits a backspace character so the receiving end can delete the text after reception.

CWinPSKView

This is a CFormView derived class that is large and contains the bulk of the user controls and convoluted logic to deal with the mouse, cursor, keyboard, and other windows functionality. This class spawns the two classes that run separate worker threads for data viewing and also the main receive and transmit algorithms. The user macro dialogs are also controlled from here. A TAB control is used to switch between different data views.

CMacrodlg

A CDialog derived class for customizing the user macros which can place user specified text, or text files, CW ID's, and automatic startup/shutoff commands into the transmit output stream.

CSquelchCtrl

A simple bar graph control that has a moveable threshold setting indicator that can be used to set the squelch threshold. The bar changes color if the input level exceeds the threshold.

CTheirCallCtrl

A CRichEditCtrl derived class that can be used to type, drag, or paste, the other stations callsign into so that the macro system can insert into the transmit output stream. It forces the text to upper case for sending.

CPlotData

This class spawns a worker thread which is used to update the data view screen area that is selected by a TAB control. The worker thread processes new incoming data and depending on the type of data view, plots it onto the screen area. This thread is a low priority thread since it is not essential to PSK31 data reception or transmitting.

Cfft

A class using code from a radix 4 FFT package written by Takuya Ooura. Very little was modified except to window the incoming data and scale the output data.

CIOCtrl

This class spawns a worker thread which instantiates either the PSK detection or Modulation class depending on the state of the Transmit/Receive button, and processes the soundcard data accordingly. A separate class, CSound is used to talk to the soundcard. A routine to twiddle a couple of pins on a serial port is used to provide a simple PTT function.

CSound

A general purpose class to either send or receive audio data from a PC soundcard. A callback function and a signaling "EVENT" is used to maintain the audio buffers.

CPSKMod

The class that takes a text character and creates the PSK31 modulated audio signal. It is only active while transmitting.

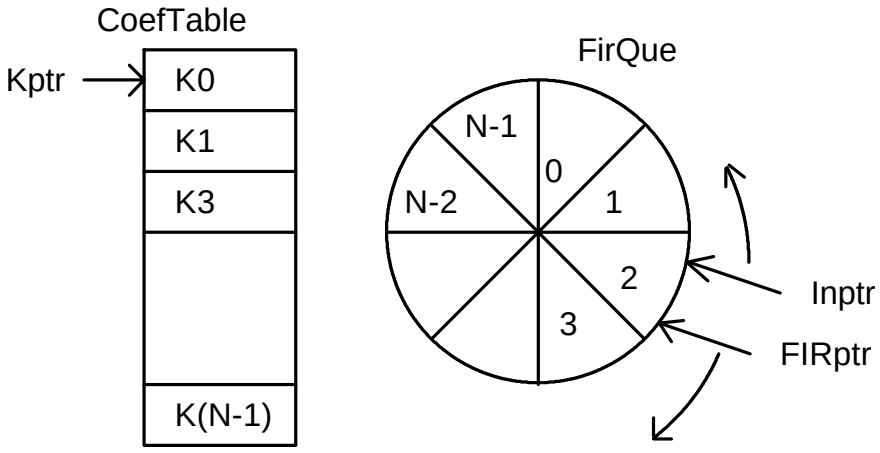
CPSKDet

The class that takes the PSK31 audio samples from the soundcard and decodes it into a text stream.

4.6. Miscellaneous Software issues

4.6.1. FIR Filter implementation

All the FIR type filters use the same structure. An array containing the FIR coefficients and a circular buffer structure with two pointers is used. New data samples are entered by decrementing the inptr and placing the new sample into the FirQue at that position. Next the FIRptr is set to the new inptr position and then the following is performed N times. For each coefficient pointed to by Kptr, the value pointed to by FIRptr is multiplied together and added to an accumulator variable. After each MAC operation, both pointers are incremented. Logic is added to deal with the circular buffer wrap around at the N-1 position in the FirQue.



The following code segment shows the general method of implementation. For a decimation type FIR filter, the MAC loop only needs to be performed at the new sample rate. For complex data, the MAC has to be performed on both real and complex parts of the data.

```
for( i = 0; i<BlockSize; i++ )        // process Blocksize new samples
{
        if( --Inptr < FirQue )        //deal with wraparound
                Inptr = FirQue +N-1;
        *Inptr = pIn[i];              // enter new sample into Que
        acc = 0.0;
        Firptr = Inptr;
        Kptr = CoefTable;
        while( Kptr < (CoefTable + N) )   //do the MAC's
        {
                acc += ( ( *Firptr++ )*( *Kptr++ ) );
                if( Firptr >= FirQue +N ) //deal with wraparound
                        Firptr = FirQue;
        }
        pOut[j++] = acc;              //save output sample
}
```

4.6.2. Inter-Class Communication

A constant aggravation with Windows MFC programming is trying to communicate or access methods between the various classes. It is easy to communicate into classes that were instantiated by the calling class. Problems arise when trying to get information back through to the parents of the class. For example the main view class can create and access classes that it creates but it is more difficult for the "children" to talk back to their parents.(probably some deep metaphor here..) The best way is to try to structure your program to minimize the need for back access to classes. If needed, one way to get around this is to pass a pointer to the parent class on to the child class as a parameter during creation. Sometimes this must be cast as a void* then recast back to the parent class to get around some problems in the order in which include files are processed by the compiler.
Another communication problem is calling MFC class methods from worker threads. Most MFC classes are not thread safe and will crash. Writing or reading text in an edit box from a worker thread will not work. One way around this is to send a windows message from the worker class to the Windows class. This method is used to transport character data to and from the DSP worker thread to the Window's edit boxes for display. The Windows messaging system can have long response times and should not be used for any high speed data traffic.

4.6.3. Processor Loading

The processor loading was measured using several methods. Windows NT has a performance monitor that can show user and kernel processing loads for various processes. Visual Studio also has a code profiler that can be used but is klunky and doesn't work very well.
Most measurements were taken using a system function that returns a 64 bit Pentium timer value that can be used to "Timestamp" any place in code.
All the timing values here and commented in the code, are based on a 133 MHz Pentium. The following table gives some average execution times of some major processing blocks.

Function	Ave time	Rate function is Called	CPU Percentage
Decimate by 2 FIR	6 mS	2.69 Hz	1.6%
Total PSK Detection Function(BPSK)	12.5 mSec	2.69 Hz	3.4%
Total PSK Detection Function(QPSK)	16.7mSec	2.69 Hz	4.5%
Decimate by 3 #1	10 uSec	18375 Hz	1.8%
Decimate by 3 #2	13 uSec	612.5 Hz	.8%
CalcBitFilter	25 uSec	612.5 Hz	1.5%
CalcAGC	9 uSec	612.5 Hz	.5%
CalcFreqError	10 uSec	612.5 Hz	.6%
SymbSync	8 uSec	612.5 Hz	.48%
DecodeSymb(BPSK)	69 uSec	31.25 Hz	.2%
DecodeSymb(QPSK)	522 uSec	31.25 Hz	1.6%
Plot Spectrum	22 mSec	2.69 Hz	5.9%
Plot Waterfall	20 mSec	2.69 Hz	5.4%
Plot Input	8 mSec	2.69 Hz	2.1%
Plot Sync	4.8 mSec	2.69 Hz	1.3%
Transmit Modulation(BPSK)	15 mSec	2.69 Hz	4%
Transmit Modulation(QPSK)	15 mSec	2.69 Hz	4%

A rough CPU % load value is displayed in the status bar of WinPSK for relative comparisons of different CPUs. It does not take into account the main process thread or any system level processing times.

Problems/Bugs/Issues

1. The receive Window scrolling functions leaves a lot to be desired. The current line many times goes out of the scroll view. This requires clicking on the scroll bar to get it back inside the view window. More hair pulling with the RichEditCtrl is needed.

2. The colors of the receive and transmit text sometimes get mixed up. See above comment.

3. The maximum likelihood detector does not appear to give any noticeable improvement in signal reception. The added complexity is probably not justified.

4. This program was written to experiment with PSK31 reception techniques and so is not a feature rich application. I have no plans to add much to it except maybe some bug fixes and minor tweaks.

5. References:

"Quoting one is plagiarism, quoting many is research"

---

[1] Windows 95/98/NT are Registered Trademarks of Microsoft Corporation
[2] Peter Martinez G3PLX. "PSK31: A new radio-teletype mode with a traditional philosophy"
[3] Peter Martinez G3PLX. "PSK31 Fundamentals"
[4] MathCad ver.6.0. MathSoft 101 Main St.,Cambridge, MA 02142
[5] Marvin E. Frerking. "Digital Signal Processing in Communication Systems"p.444. ISBN0-442-01616-6
[6] George B. Thomas, Jr. "Calculus and Analytic Geometry" p.238
[7] W.T. Webb and L. Hanzo "Modern Quadrature Amplitude Modulation" p.367 ISBN0-7273-1701-6
[8] Yuri Okunev. "Phase and Phase Difference Modulation in Digital Communications" p.173-216 ISBN 0-89006-937-9
[9] Bernard Sklar. "Digital Communications Fundamentals and Applications" ISBN 0-13-211939-0
[10] "C. Britton Rorabaugh. "Error Coding Cookbook". P. 127 ISBN 0-07-911720-1
[11] Tom McDermott, N5EG. "Wireless Digital Communications: Design and Theory" ISBN 0-9644707-2-1
[12] Peter Martinez G3PLX. "Description of the Half-Rate QPSK code proposed for the QPSK/FEC Extension to PSK31"