SSTV1 Modem Technical Description

Project Description

The SSTV modem software project was an experiment in using DSP methods to modulate and demodulate an audio FM Slow Scan TV signal using the TAPR/AMSAT DSP-93 platform. Traditional decoders and encoders use either a simple op-amp limiter, or PC sound card to decode the FM audio SSTV signal. This project explored the use of a digital phase locked loop to perform the demodulation function.

Some of the SSTV1 modem implementation features:

- AGC function on input samples.
- I and Q quadrature signal components generated with Hilbert Bandpass filters.
- Digital complex PLL used for obtaining input frequency information.
- SSTV encoding using a phase coherent modulation of a numerically controlled oscillator.
- Automatic CW callsign ID after picture transmission.
- Communication with modem uses 38400bps serial data using DSP-93 UART.
- Uses JVFAX(7.0 or 7.1) for user interface and various picture format conversions.

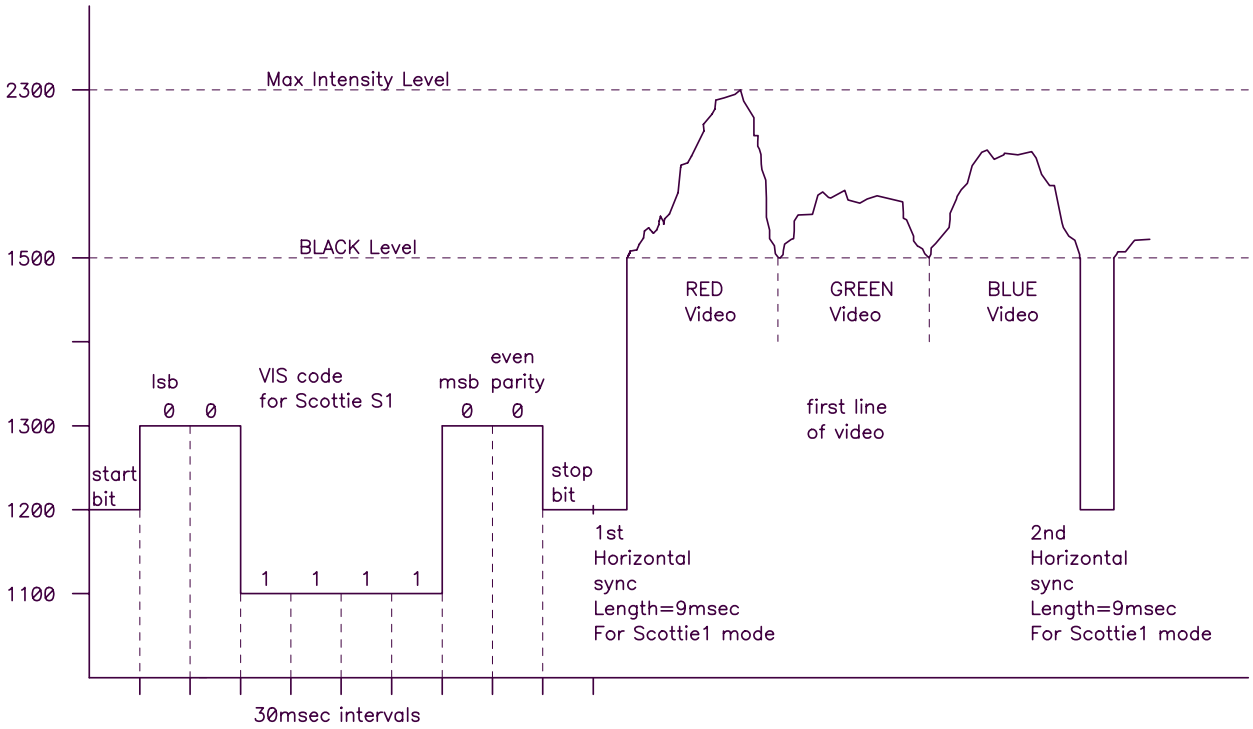
Slow Scan Television(SSTV) is an old mode that was developed to send low frame rate B/W television images over standard voice bandwidth audio channels. Due to the proliferation of low cost PC's, this basic modulation scheme has evolved into a method of transferring full color still pictures stored on PC's back and forth between stations. Although a zillion different picture transmission modes exist, only a handful are popular and the low level modulation scheme is common to them all. This project addresses the low level decoding and encoding of this low level audio FM signal and lets the high level picture protocol and synchronization be handled by a sort of shareware without a fee program, JVFAX.

The SSTV Signal

The first step is to understand a little about the nature of the SSTV signal. Information on the details of SSTV signals is sketchy at best. The information here is based on various sources found on the Internet and also in descriptions given in the documentation found with the various SSTV programs. The accuracy of the information is therefore somewhat uncertain. Because of the large number of modes we'll only look at the details of one of the more popular ones called "Scottie-S1" developed by E.T.J. Murphy GM3SBC.

The beginning of a transmission consists of a vertical sync pulse that is a 300msec interval of 1200 Hz tone. (Usually there is a period of tone prior to the VIS code but I haven't found any details or standards for this pre VIS tone) Most programs also encode on top of this vertical sync pulse a serial data stream called a VIS(Vertical Interval Signaling) code that can be decoded and used to specify which transmission

format is being sent. The VIS code uses 1100 and 1300Hz tones to indicate “1”’s and “0”’s respectively in a serial data byte. The following diagram illustrates the beginning of a picture and the first line of image data.



Note that each line of video is divided into three color sections Red, Green, and Blue. The picture decoder software must overlay these three sections together to form the line of a full color image. The horizontal sync pulse is used to indicate the beginning of a new line of picture data. Usually once the first few horizontal sync pulses are found, the decoding software ignores them and relies on accurate internal time references to ride through the rest of the picture. This requires accurate calibration of the software to prevent slanted pictures. Most software packages provide some means to adjust their internal clocks to compensate for clock inaccuracies.

Scottie-S1 mode consists of 256 lines of video data. The time between each horizontal sync pulse is 429mSec. This gives a total transmission time of .3+256*.429=110.1 seconds.

The number of picture elements within each line is a function of how fast one samples the video data and the bandwidth of the video data signal. JVFAX displays 340 pixels per line while most other programs display 320. The worst case scenario is a picture consisting of 340 alternating lines of black and white. A test of several of SSTV programs revealed that in reality only about half this number of lines are actually able to be transmitted. This would mean that the SSTV signal has to make 170 transitions

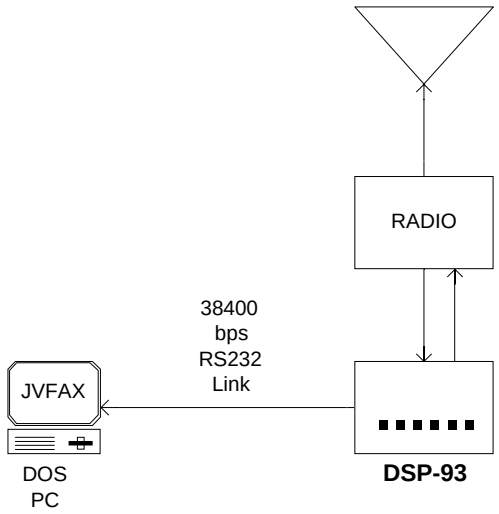
from 1500 to 2300Hz in a period of .429/3=.140 seconds. This represents a modulation frequency of (170/2)/.17=500Hz. The peak deviation is (2300-1500)/2=400Hz.

The modulation index(β) is the peak deviation/modulation frequency = 400/500 = .8. For tone modulation the FM signal bandwidth is about 2(β+1)*modulation frequency. So the FM bandwidth is about 2*(.8+1)*500 = 1800Hz.

In reality, bandwidth is traded off for better noise and weak signal operation. After a lot of on the air testing it was found that a good trade off of picture resolution and noise performance was with a 1300Hz filter bandwidth from 1050Hz to 2350Hz.

The DSP-93 platform consists of a 40 MHz Texas Instruments TMS320C25 16 bit DSP chip, surrounded by 32K words of program memory and 32K words of data RAM. An analog interface board contains a TI TLC32044 14 bit A/D, D/A converter, a asynchronous UART chip, and various I/O ports for radio control and LED display control. A software controllable gain block is provided for adjusting the receiver input level to the A/D converter. A monitor EPROM is used to provide a downloader function as well as storing built in modem software and test applications. Programs can be downloaded into the DSP-93 using utility programs that run on a PC.

The following diagram shows the basic setup for using the DSP-93 to send and receive SSTV signals. A DOS based PC running the program JVFAX(7.0 or 7.1) is the user interface. The DSP-93 connects to the radio using one of the radio ports. It connects to the PC using a standard COM serial port configured to run at 38400 bps. A little more information is given in the file SSTV1USR.PDF concerning setup of JVFAX to operate in this mode.



Software Design Method

The software for the SSTV1 modem was designed in a modular fashion using “C” language blocks to describe each function. Once the code blocks were defined, then the C320-25 assembly code was hand assembled using the “C” code blocks as a reference. This may seem cumbersome but designing in assembly language can get very complicated and confusing in a hurry even with generous comments. By designing the code in a higher level language, one doesn’t get bogged down in implementation details until the design is ironed out. This may take a little longer to get to the debug stage, but reduces the number of bugs once you get there, especially as the program gets more complicated.

The software source was also broken into several parts mainly to ease in editing. This sort of implements a “poor man’s” linking assembler in that one can edit and debug using just the file associated with a general task instead of having to search through one large cumbersome source file. When assembling of course, all the files have to be re-assembled.

Data queues(FIFO, Circular, or “rubber band” buffer) are used on the A/D and D/A channel to reduce the timing constrains on the software and allow even distribution of processing time.

The only time critical operation is the actual A/D and D/A sampling operation which is performed by the TLC32044 CODEC chip in conjunction with the DSP hardware. Since the data is taken from or put into the data sample queues at precise time intervals, the rest of the software is not constrained to operate on the data in real time. This allows the software to perform periodic operations longer than the sample time interval as long as the average processing time does not exceed the sample time interval.

Another software method used was the use of indirect function calls to implement state machines which are of use at various places in the code. Basically the address of the function to call is placed in a RAM variable. An indirect call using that RAM variable results in execution of the specified function. Within that function, a “NEXT STATE” can be specified by simply loading the RAM variable with the address of the next state function. In this manner, complicated state machines can be implemented fairly easily.

The software is broken into eight files for easier manipulation:

- SSTVMAIN.ASM This is the main entry file and is the one that is specified for assembling as it has all the include references for the auxiliary files. It contains Constant definitions, variable allocations, hardware and software initialization, interrupt service routines, and low level bit twiddling functions. The main code service loop also resides here which calls all the other modules in a round robin fashion to service all the various tasks of the modem.

- SSTVDATA.TBL This file contains various constant data tables used throughout the modem. A SIN table, a CW lookup table, and FIR coefficients are contained here.

- SSTVAIN.ASM This file contains all the routines that service the A/D input samples as they arrive and demodulate it.

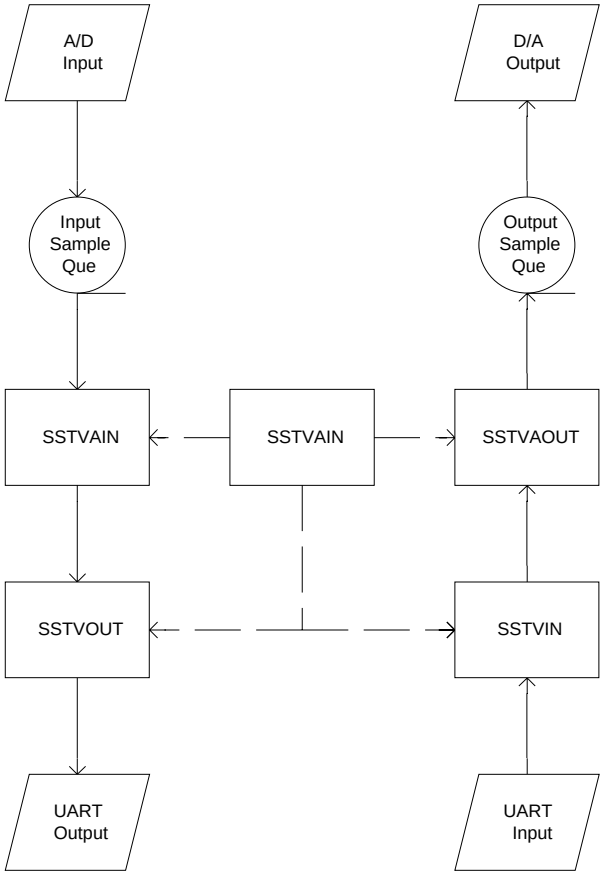
- SSTVAOUT.ASM This file contains all the routines that create the SSTV audio signal for transmission. The D/A output samples are generated here.

- SSTVIN.ASM This file contains all the routines that service the JVFAX formatted input bytes as they arrive from the DSP-93 UART and creates the proper frequency data for the SSTVAOUT module.

- SSTVOUT.ASM This file contains all the routines that generate the JVFAX formatted serial output data from the demodulated frequency data as it is decoded by the SSTVAIN module.

Software Descriptions

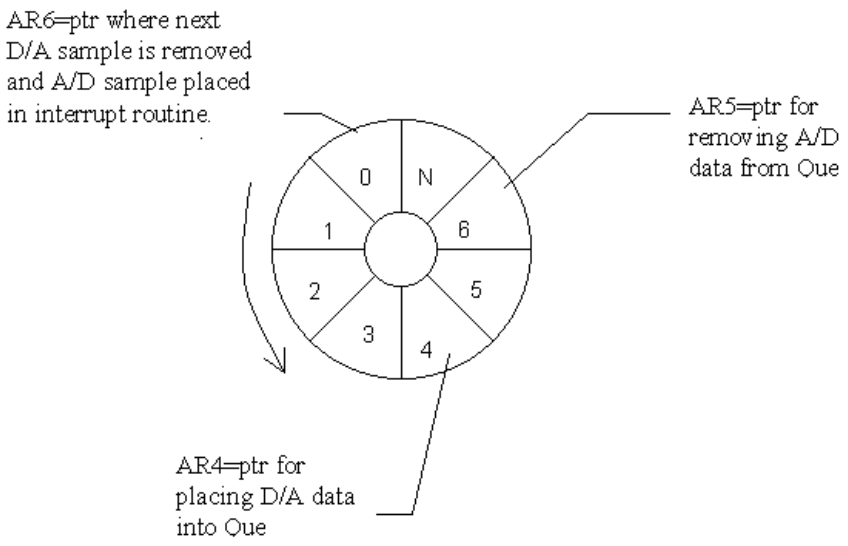
The software begins executing after being downloaded by first initializing the hardware resources on the DSP-93. The TLC32044 AIO chip is initialized to run at a sample rate of 10893 sample per second. The 16C550 UART chip is initialized to 34800 bps. This is the JVFAX interface rate. The onboard timer of the TMS320C25 chip is set to interrupt every 5 milliseconds. This is used as a general purpose timer for some of the initialization routines, LED display, and CW ID timing functions. Several initialization routines are called to initialize various variables used by each module. After initialization, the interrupts are enabled, and the main service loop is entered in which all four software modules are called in a loop continuously.



First the AIO interrupt service routine will be described. It's function is to send a new sample from the Sample_Que out the D/A PORT and store a new A/D sample into the Sample_Que. Since the A/D and D/A are run at the same sample rate, only one interrupt service routine is used for both. Also since one word is removed and one word is placed in the Sample_Que at every sample time, only 3 pointers are need to maintain the Sample_Que.

```
void RxIntService(){
   Save_Context();
   DXR = Sample_Que[AR6];              // write D/A from Sample_Que
   Sample_Que[AR6++] = DRR;            // read A/D into Sample_Que
   if( AR6>Sample_Que+SAMPQUESIZE-1 )  // deal with wrap around
      AR6 = Sample_Que;
   Restore_Context();
}
```

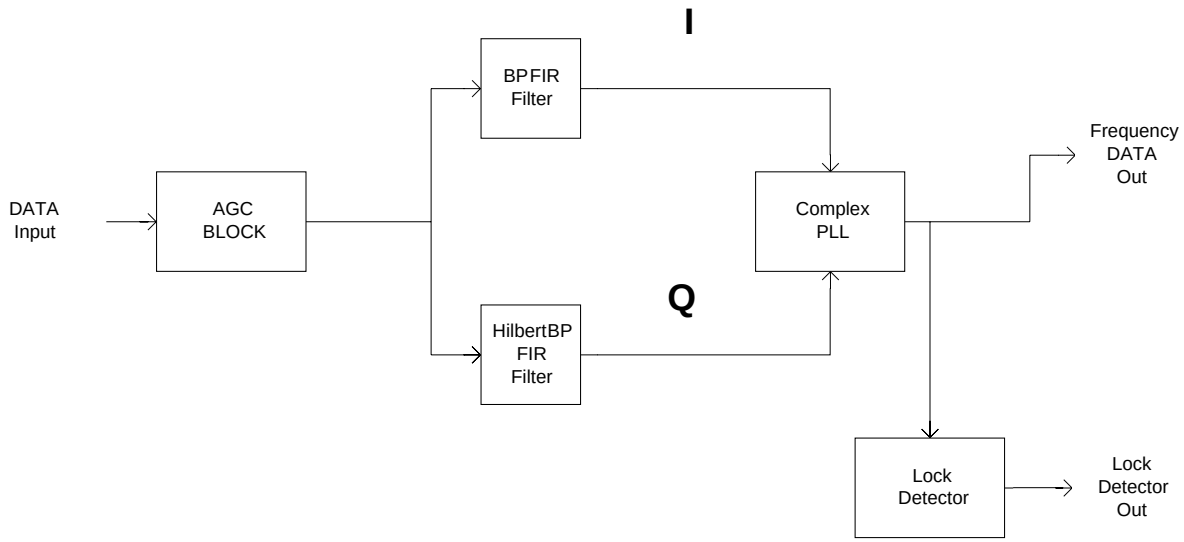
Three Auxiliary registers(AR6,AR5,AR4) are used as pointers to the Sample_Que. AR3 is used as a software stack pointer to save and restore processor context since the 320C25 doesn't save anything except the return address during interrupts.



A/D data can be removed from the Sample_Que as long as AR5 != AR6.
D/A data can be placed in the Sample_Que as long as AR4 != AR5.

SSTVAIN.ASM Module

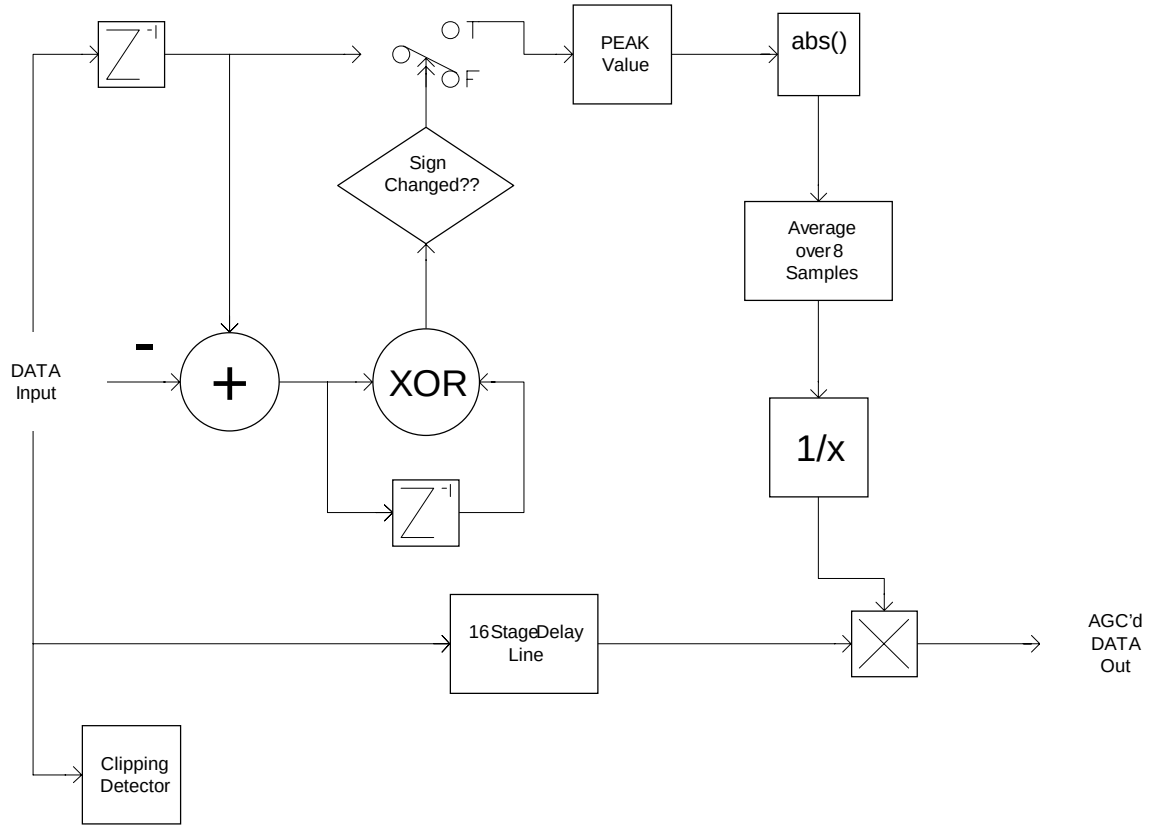
The block diagram below shows the general operation of this module. First the samples are processed by the AGC block to try and keep the amplitude constant. The signal is then bandpass filtered and split into I and Q terms and fed to the complex PLL block. The PLL outputs a signal proportional to the input signal frequency and also a lock indicator to qualify the output as valid frequency data.



A/D samples are pulled out of the Sample_Que and passed through a clipping detector. This block just sees if any samples are above some peak threshold and flashes a front panel LED. This is useful in setting the maximum receive audio level.

The AGC works by monitoring the incoming sample stream and calculating the slope of the incoming signal by subtracting the present sample from the previous sample. If the sign of this difference(slope) changes, then a peak in the incoming wave form has occurred. The absolute value of the sample is then stored as the signal peak level. A running average is calculated over 8 samples to smooth out the signal peak values. A 16 stage delayed version of the input is then multiplied by the inverse of the average peak signal to obtain an AGC'd signal for the remaining signal processing.

The delay line enables one to act on "future" input data and begin the AGC action on the signal before it arrives at the end of the delay line. An input spike can be detected by the peak detector and the gain can be reduced before the spike actually reaches the end of the delay line and disrupt downstream processing. LED2 is turned on if the input signal is below the level where the AGC can operate.



I/Q Signal Generation

The next step is to band pass filter the AGC'ed input samples and split them into their I/Q components. This could be done using a mixing scheme but another method was chosen. A band pass filter pair can be generated using two FIR filters that not only provide the desired band pass characteristics but also creates the complex outputs. The filters are calculated using the following method as described in Frerking's book "Digital Signal Processing in Communication Systems" and I'm sure in others too.

1. Create a low pass FIR filter with the desired pass band and stop band characteristics assuming that "0" frequency is the desired band pass center frequency.

2. Transform the low pass coefficients into the I and Q band pass filters using the following conversion:

$$Ih_{BP}(n) = 2h_{LP}(n)\cos(2\pi f_C[n - \frac{N-1}{2}]T)$$

$$Qh_{BP}(n) = 2h_{LP}(n)\sin(2\pi f_C[n - \frac{N-1}{2}]T)$$

Where $h_{lp}(n)$ = FIR coefficients of the low pass filter.
Fc = Band pass center frequency
N = number of FIR taps
T =sample period
$Ih_{BP}(n)$ = FIR coefficients for the I filter
$Qh_{BP}(n)$ = FIR coefficients for the Q filter

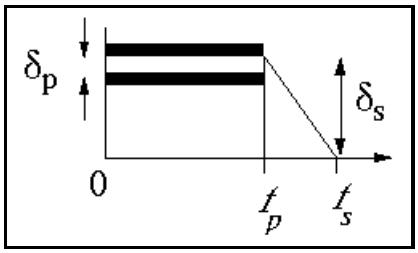
The following C program converts a file containing the low pass filter coefficients into two files containing the I and Q bandpass FIR filter coefficients:

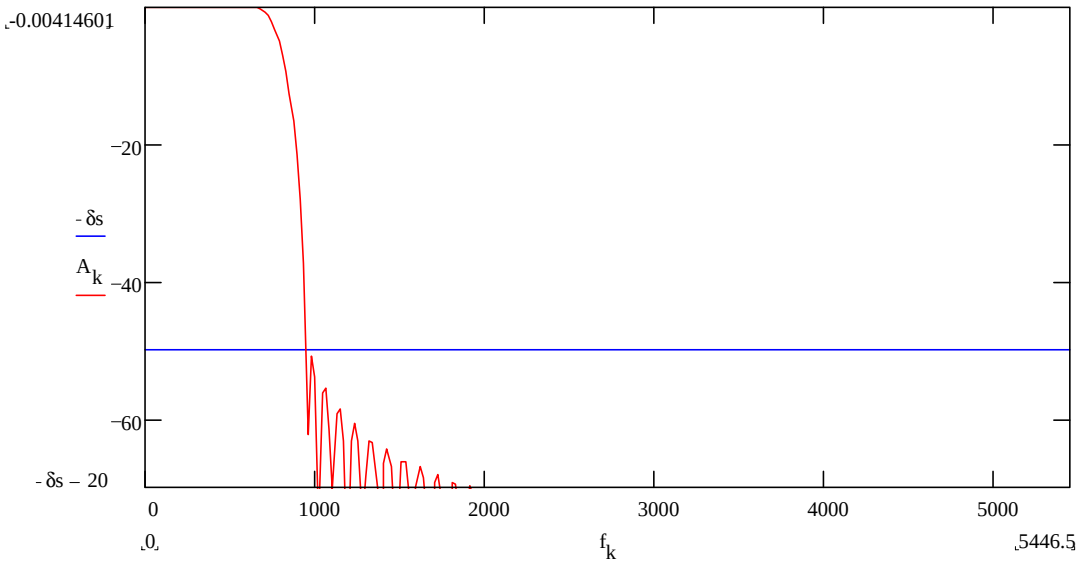
```
/*=========================================================================*/
/*   Low pass FIR Coeficient conversion to Hlbert bandpass                 */
/*         by M. Wheatley  03-01-1997                                      */
/* Last Changed < lptobp.c >  Saturday 3-1-1997    8:32 AM                 */
/*.........................................................................*/
/*   This program takes as a command line argument the name of a file       */
/*containing the coefficients of a low pass FIR filter in ASCII format.     */
/*   It then asks for the sample frequency, and the desired bandpass center  */
/*frequency.                                                               */
/*      The program outputs two files with the same name as the name given in*/
/*the command line argument but with the extensions .ibp and .qbp denoting   */
/*the I and Q band pass filter coefficient files.                          */
/*=========================================================================*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#define MAXTAPS 255

FILE *strmin;
FILE *strmout1;
FILE *strmout2;
char string[512];
char * newstr;
char outfile1[128];
char outfile2[128];

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
/*                             ---------                                   */
/*                             | m a i n |                                 */
/*                             ---------                                   */
/* This routine is called when the program starts.                         */
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void main( int argc, char *argv[] )
{
double hlp, ibp, qbp, cfreq, Fs, Wo;
int linecount, N;
int ix;
   N = argc;
   ix = 0;
// get input file name
   while( argv[1][ix] != 0 && argv[1][ix] != '.' ){
      outfile1[ix] = argv[1][ix];
      outfile2[ix] = argv[1][ix];
      ix++;
   }
   outfile1[ix] = '.';  outfile2[ix++] = '.';
   outfile1[ix] = 'i';  outfile2[ix++] = 'q';
   outfile1[ix] = 'b';  outfile2[ix++] = 'b';
   outfile1[ix] = 'p';  outfile2[ix++] = 'p';
   outfile1[ix] = 0; outfile2[ix++] = 0;

   printf( "\nEnter Sample frequency (Hz)-");
       ix = scanf("%lf", &Fs );
   printf("\n Sample Freq = %8.4f",Fs );
   printf( "\nEnter Bandpass Center Frequency (Hz)-");
```

```
        ix = scanf("%lf", &cfreq );
   printf("\n Center Freq = %8.4f",cfreq );
   Wo = 2 * 3.14159 * cfreq;
/*-------------------------------------------------------------------------*/
/*   See if can open specified file for reading & writing                  */
/*-------------------------------------------------------------------------*/

   if( (strmin = fopen(argv[1],"r")) != NULL &&
      (strmout1 = fopen(outfile1,"w")) != NULL &&
      (strmout2 = fopen(outfile2,"w")) != NULL )   {
         printf("\nFile [ %s ] is being processed.\n", argv[1]);
      N = 0;
// find filter length
      do{
         newstr = fgets(string,MAXTAPS-1,strmin);
         if( newstr != NULL ){
            N++;
         }
      }while( newstr != NULL );
      printf("\n Filter Length = %d\n\n",N );

      rewind( strmin );
/*-------------------------------------------------------------------------*/
/*   Read file into buffer one line at a time, till the                    */
/*   the EOF is reached.                                                   */
       linecount = 0;
       do{
          newstr = fgets(string,MAXTAPS-1,strmin);
          if( newstr != NULL ){
             hlp = atof( newstr );     // convert coeficient string  to float
// apply transform to low pass filter coefficients
             ibp = 2.0 * hlp * cos( (Wo/Fs)*(linecount-( (N-1)/2 ) ) );
             qbp = 2.0 * hlp * sin( (Wo/Fs)*(linecount-( (N-1)/2 ) ) );

             fprintf(strmout1,"%8.6f\n",ibp );
             fprintf(strmout2,"%8.6f\n",qbp );
             linecount++;
          }
       }while( newstr != NULL );
       printf("\nConverted %d coefficients.\n", linecount);
       printf("\nOutput files are [ %s ], [ %s ]", outfile1, outfile2 );
                fclose(strmin);
                fclose(strmout1);
                fclose(strmout2);
   }
/*-------------------------------------------------------------------------*/
/* Here if can't open the specified input/output files
       */
/*-------------------------------------------------------------------------*/
   else {
      printf("\n\nSpecified file(s) cannot be opened.\n");
   }
   exit(0);
}
/*++++++++++++++++++++   E N D   P R O G R A M   ++++++++++++++++++++++++++*/
```

The low pass filter coefficients were obtained using a "MathCAD" program. The design parameters for the SSTV lowpass filter were

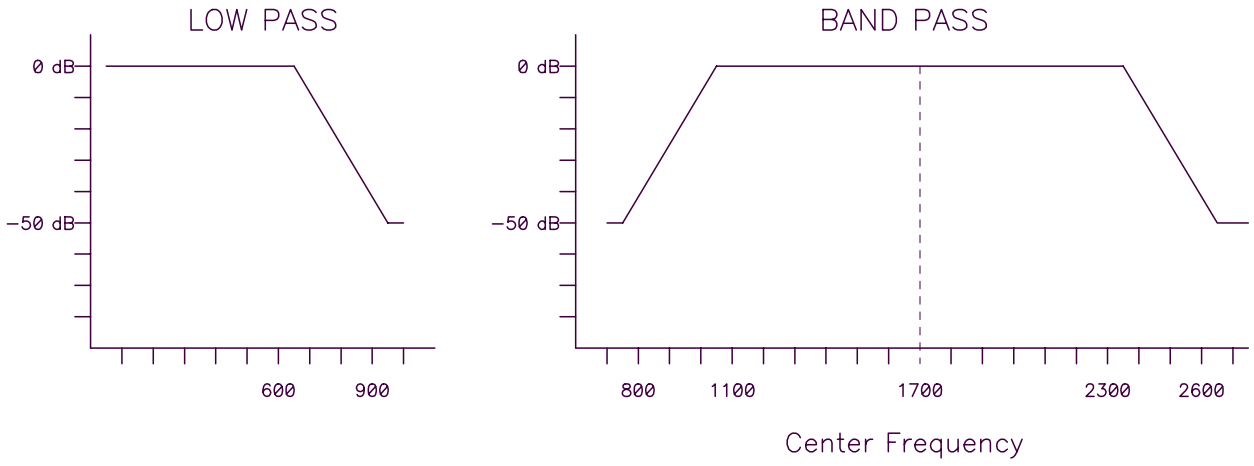
Filter Specification:
Sampling Frequency in Hz: = 10893
Passband Ripple in dB (Peak Gain=1) = 1dB
Passband Frequency in Hz = 650 Hz
Stop Band Attenuation in dB = 50dB
Stop Band Frequency in Hz = 950 Hz

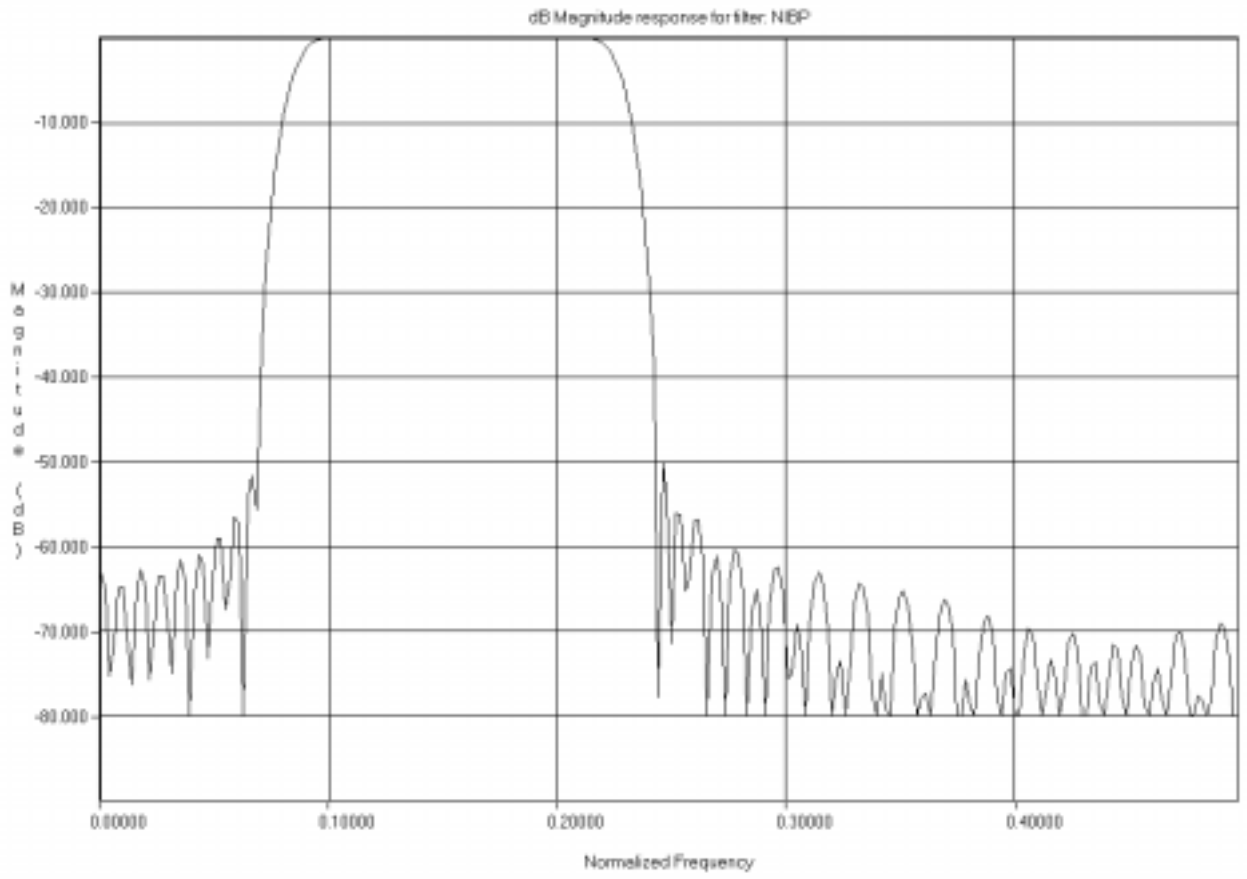




The above is the Mathcad low pass filter response.

The low pass filter was transformed into two band pass filters by multiplying the LP coefficients by the earlier described equations. The resulting filters were two 107 Tap FIR filters to generate the I and Q signals.





Above is a plot of the I channel band pass FIR filter. The frequency scale is normalized to the sample frequency of 10893Hz.

Complex PLL Detector

The next task is to use the I/Q input signals to phase lock a complex NCO (numerically controlled oscillator). The resulting frequency feedback control signal can then be used as the demodulated FM output signal.

The input signal is complex and can be written as:
$Ae^{-j\varpi_{in}t} = A\cos(\varpi_{in}t) + jA\sin(\varpi_{in}t)$

The numerically controlled oscillator signal is also complex and can be written as:
$e^{-j\varpi_{nco}t} = \cos(\varpi_{nco}t) + j\sin(\varpi_{nco}t)$

Multiplying the input signal by the complex conjugate of the NCO signal results in the following terms:

$$Ae^{-j\varpi_{in}t}e^{j\varpi_{nco}t} = Ae^{-j(\varpi_{in}t-\varpi_{nco}t)} = A\cos(\varpi_{in}t-\varpi_{nco}t) + jA\sin(\varpi_{in}t-\varpi_{nco}t)$$

First look at the imaginary term $A\sin(\varpi_{in}t-\varpi_{nco}t)$.

The argument of the sin() term is the phase difference between the incoming signal and the locally generated NCO. This is the phase error term that can be used as feedback to the PLL. Once locked, the phase error will be small so $A\Delta\theta \cong A\sin(\Delta\theta)$.

Note the amplitude of the input signal "A" is part of the Phase Error term and in order to keep the phase error gain constant in the PLL, the input signal must be kept at a constant amplitude thus the need for AGC before the detection process.

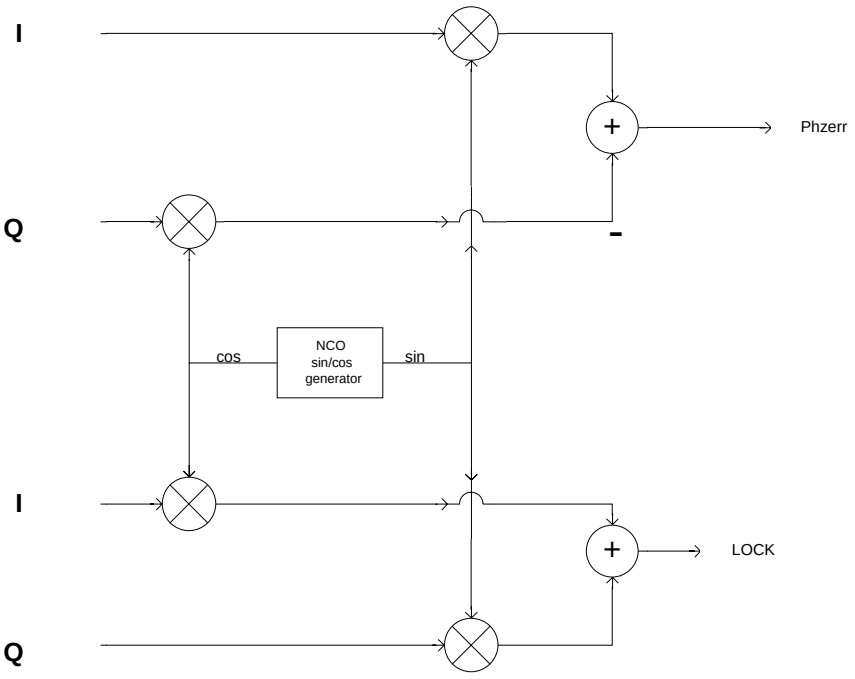
Dusting off an old math book revealed the following trig identity:
$\sin(x-y) = \sin(x)\cos(y) - \cos(x)\sin(y)$

Applying it to the term $A\sin(\varpi_{in}t-\varpi_{nco}t)$ gives:
$Phzerr \cong A\sin(\varpi_{in}t-\varpi_{nco}t) = A\sin(\varpi_{in}t)\cos(\varpi_{nco}t) - A\cos(\varpi_{in}t)\sin(\varpi_{nco}t)$

Since $A\cos(\varpi_{in}t)$ =real part of input signal or the Iterm
and $A\sin(\varpi_{in}t)$ =imaginary part of input signal or the Qterm,

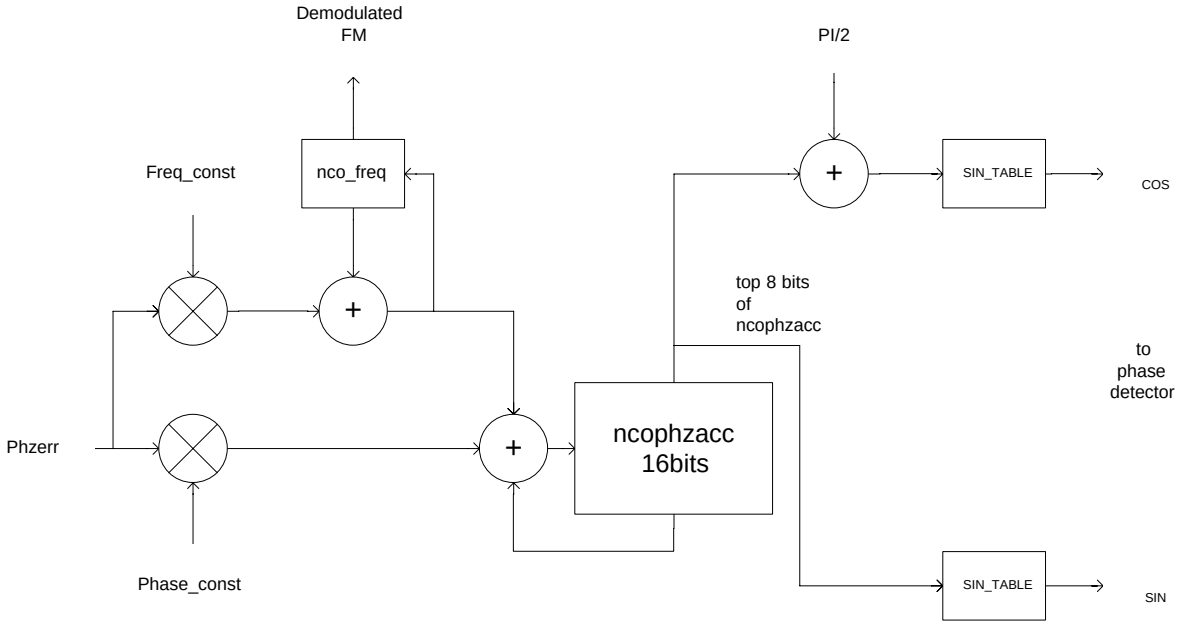
$$Phzerr \cong (Qterm)\cos(\varpi_{nco}t) - (Iterm)\sin(\varpi_{nco}t)$$

Next look at the real term of the complex conjugate multiplied input signal term $A\cos(\varpi_{in}t-\varpi_{nco}t)$.
If the phase difference is zero(PLL is locked) then $\varpi_{in}t = \varpi_{nco}t$
Thus $\cos(\varpi_{in}t-\varpi_{nco}t) = \cos(0) = 1$. Thus this term is at a maximum when the phase error is zero. This term can be used to help determine when phase lock has occurred.
The following is a block representation of the phase error and lock detector.



Numerically Controlled Oscillator and PLL

The NCO section is described by the following block diagram:



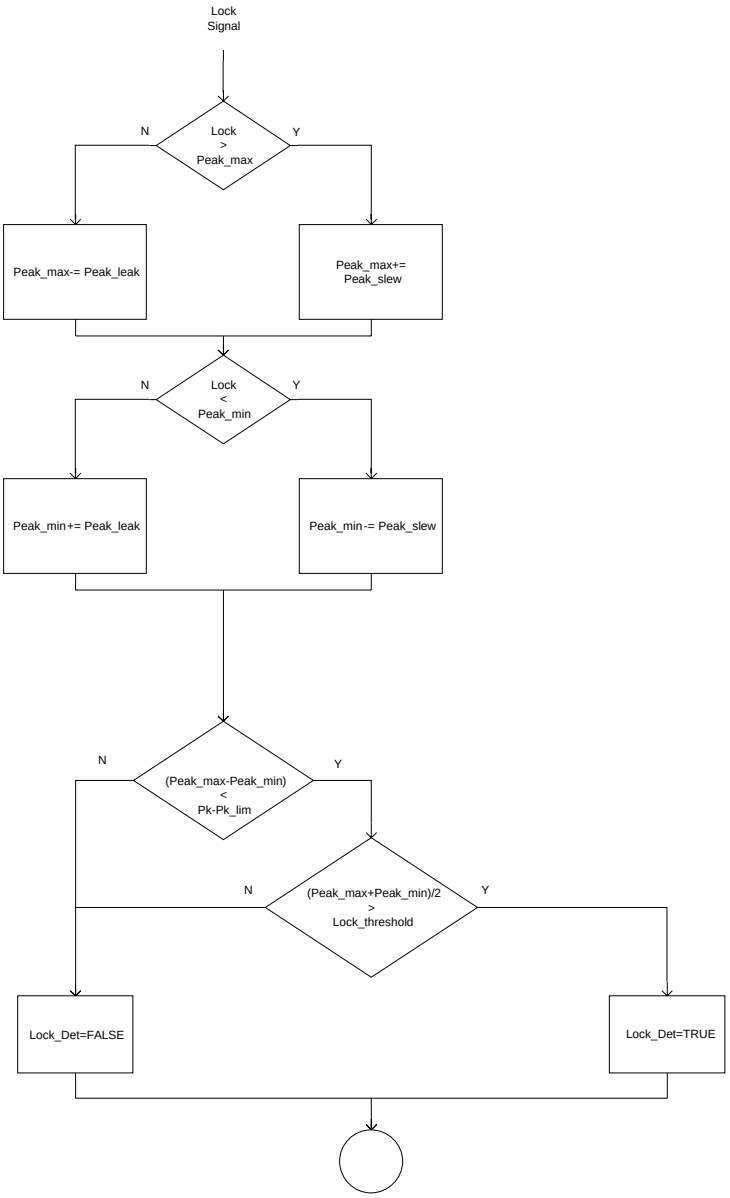
The NCO uses a phase accumulator that increments by an amount depending on the phase error and also the value of “nco_freq” which is a value that is the integral of the

phase error and represents the NCO frequency. If one looks at the process when phase locked, the phase error will be zero so the ncophzacc will be incremented each sample time by the value of nco_freq. Two constants, Freq_const and Phase_const are used to set the closed loop characteristics. In general, the Phase_const controls the loop damping while the Freq_const controls the overall gain or “stiffness” of the control loop. The values are not orthogonal and each affects the other.

Not shown in the block diagram is a clamping function which limits the range of nco_freq so it does not get too far out of the operating range when the PLL is not locked.

PLL Lock Detection

The last function to be described is the PLL lock detector. The basic idea is to look at the lock signal generated by the phase detector section and keep track of the maximums and minimums using two leaky integrators. If the difference between the min and max is less than some threshold and the average of the max and min is above a certain threshold, then the PLL is said to be locked. The first condition takes care of the case where noise is the predominate signal and the PLL cannot lock onto the noise. The second test condition takes care of the case where the signal and noise are so low that there is not enough signal to cause phase lock. The following diagram shows the basic lock detector logic flow.



The following are code snippets from the SSTVAIN.ASM module showing the AGC and DPLL processes.

```
;=============================================================================
;             =======   A G C _ c a l c ( Temp1 )     =======
;=============================================================================
;This routine is called every A/D sample and calculates a new AGC gain value.
;
;void AGC_calc(Temp1)
;   DeltaNew = OldInput - NewInput;
;   if( ( DeltaNew ^ DeltaOld ) & BIT15 ){      // if sign changed
;      OldInput = abs( OldInput );      // OldInput is peak signal
;      AGC_Sum = AGC_Sum + OldInput - AGC_Ave;
;      AGC_Ave = AGC_Ave/8;
;   }
;   DeltaOld = DeltaNew;
;   OldInput = Temp1;
;
```

```
;   AGCdelay[ 0 ] = Temp1;              // put latest sample in delay line
;   Temp1 = AGCdelay[ AGC_LENGTH-1 ];   // get oldest sample out for processing
;   for( i=AGC_LENGTH-2; i >= 0; i-- ){ // shift everybody down
;      AGCdelay[ i+1 ] = AGCdelay[ i ];
;   }
;   Temp1 = Temp1/AGC_Ave;              // scale sample by AGC gain
;}


;=============================================================================
;      =======   P r o c e s s I n p u t ( input )     =======
;=============================================================================
;This routine is called every A/D sample and processes the AGC'ed input
;sample.  The input is bandpass filtered into two channels(I andQ) using,
;a normal bandpass filter for the Iterm and an identical bandpass filter with
;a hilbert transform for the Qterm.  A complex nco is implemented with a
;phase accumulator(ncophzacc) that is added to by a phase error(Phzerr) term
;and a frequency term (nco_freq).
;
;void ProcessInput( input )
;{
;   BPFdelay[ 0 ] = input;      // do FIR calculations to get I and Q
;   acc = 0;
;   for( i=BPFIR_LENGTH-1; i >= 0; i-- ){
;      acc += BPFdelay[ i ] * IBPFIRCOEF[ i ];
;   }
;   Iterm = acc;                // Iterm =  I bandpass output data
;   for( i=BPFIR_LENGTH-1; i >= 0; i-- ){
;      acc += BPFdelay[ i ] * QBPFIRCOEF[ i ];
;      BPFdelay[ i+1 ] = BPFdelay[ i ];         // do the shuffle
;   }
;   Qterm = acc;                // Qterm =  Q hilbert bandpass output data
;
;   ncophzacc = ncophzacc + nco_freq + Phzerr;   // update new phase
;   nco_sin = SIN_TABLE[ ncophzacc>>8 ];        // create nco sin and cos
;   nco_cos = SIN_TABLE[ ncophzacc>>8 + PHASE90 ];
;
;   Lock = Iterm*nco_cos + Qterm*nco_sin;       // calc lock signal
;   Check_lock();                               // see if phaselocked
;
;   Phzerr = Qterm*nco_cos - Iterm*nco_sin;     // calc phase error
;   Phzerr = Phzerr * Phase_const;              // scale phase error
;
;   nco_freq =  nco_freq + (Phzerr * Freq_const); // calc new frequency
;                                               // for phase lock
;   if( nco_freq < MIN_freq )          // clamp new frequency within bounds
;      nco_freq =  MIN_freq;
;   else
;      if( nco_freq > MAX_freq )
;         nco_freq =  MAX_freq;
;}
;.............................................................................
;=============================================================================
;             =======   C h e c k _ l o c k  ( Temp1 )     =======
;=============================================================================
;This routine is called to determine if the PLL is locked. The peak to peak
;value of a signal is found and if the pk-pk value is less than a specified
;limit and the overall value is above a specified limit then the PLL is
;locked. The input sample is in "Temp0"
;
;void Check_lock( Temp0 )
;   if( Temp0 > Peak_max )      // if greater than current max level
;      Peak_max += Peak_slew;
;   else
;      Peak_max -= Peak_leak;
;   if( Temp0 < Peak_min )      // if less than current min level
;      Peak_min -= Peak_slew;
;   else
;      Peak_min += Peak_leak;
;   if( Peak_max > Peak_max_lim )      // clamp to peak limits
;      Peak_max = Peak_max_lim;
;   if( Peak_min < Peak_min_lim )      // clamp to peak limits
;      Peak_min = Peak_min_lim;
;
;   Temp0 = Peak_max - Peak_min;
```

```
;   Temp1 = (Peak_max + Peak_min) / 2;
;   if( (Temp0 <= Pk_Pk_lim) && (Temp1 >= Lock_thresh) ){
;      Flags.LOCK_DET = TRUE;
;   }
;   else{
;      Flags.LOCK_DET = FALSE;
;   }
;
;}
```

SSTVOUT.ASM Module

This module uses the demodulated FM value from the SSTVAIN.ASM module and generates the proper stream of UART serial data bytes for use by JVFAX in decoding the SSTV pictures.

JVFAX has specified a data format for serial input devices to send SSTV information for decoding.

The first two bits of each data byte indicate the possible sync frequency ranges.

```
Bit 1 0     identifies:
=======================
    H L     1050-1100 Hz VIS “1”bit
    L L     1150-1250 Hz HSYNC,VSYNC
    L H     1250-1350 Hz VIS “0”bit
    H H     1500-2300 Hz video data
```

The upper 6 bits contain video information where 0 = 1500Hz and 63 = 2300Hz. This provides 64 levels of intensity information to the JVFAX program.

The DSP-93 UART continuously sends data bytes to JVFAX at a rate of 38,400 bps which equates to a video sample rate of 3840Hz. Sync signals need to be a little more robust since they determine the start of a picture and each line. For this reason, the PLL lock signal is used to qualify any signal in the sync range so that false starts will be reduced. Image video data on the other hand is allowed to pass through regardless of the lock state because it was thought that noise in an image looked better than complete dropouts of the image during signal fades. It would be better if another state could be sent to the PC program to tell it that the signal was invalid and let it intelligently “fill in” the areas that were noisy.

Another function of this module is to flash three LED’s to aid in tuning.
LED213 flashes when a signal is in the 1050-1150 Hz range.
LED214 flashes when a signal is in the 1150-1250 Hz range.
LED215 flashes when a signal is in the 1250-1350 Hz range.

During SSTV reception, LED214 should flash every .429 seconds during the Horizontal sync pulse if the receiver is tuned on frequency. LED213 indicates its too low and LED215 indicates it’s too high. (This is assuming the signal is an SSB signal. If an FM or AM signal is being received then the signal should be tuned for maximum amplitude minimum distortion.)

SSTVIN.ASM Module

This module gets incoming DSP-93 UART data bytes from the JVFAX program during transmit and converts them into the proper format to be converted to tones by the SSTVAOUT.ASM module. The Push To Talk(PTT) is generated to key a transmitter based on the continuous reception of UART data bytes from JVFAX. If at least some number of bytes are received per second, then it is assumed that a transmission is taking place. When the bytes are no longer being received, it is assumed the picture transmission is done and a state machine is activated that sends out a CW ID string by amplitude modulating the tone generator at a fixed frequency.

Strangely, JVFAX has a different data format for picture transmission than for reception. For transmitting, three unique codes are used to signify the sync frequencies.
125 = 1100Hz
126 = 1200Hz
127 = 1300Hz

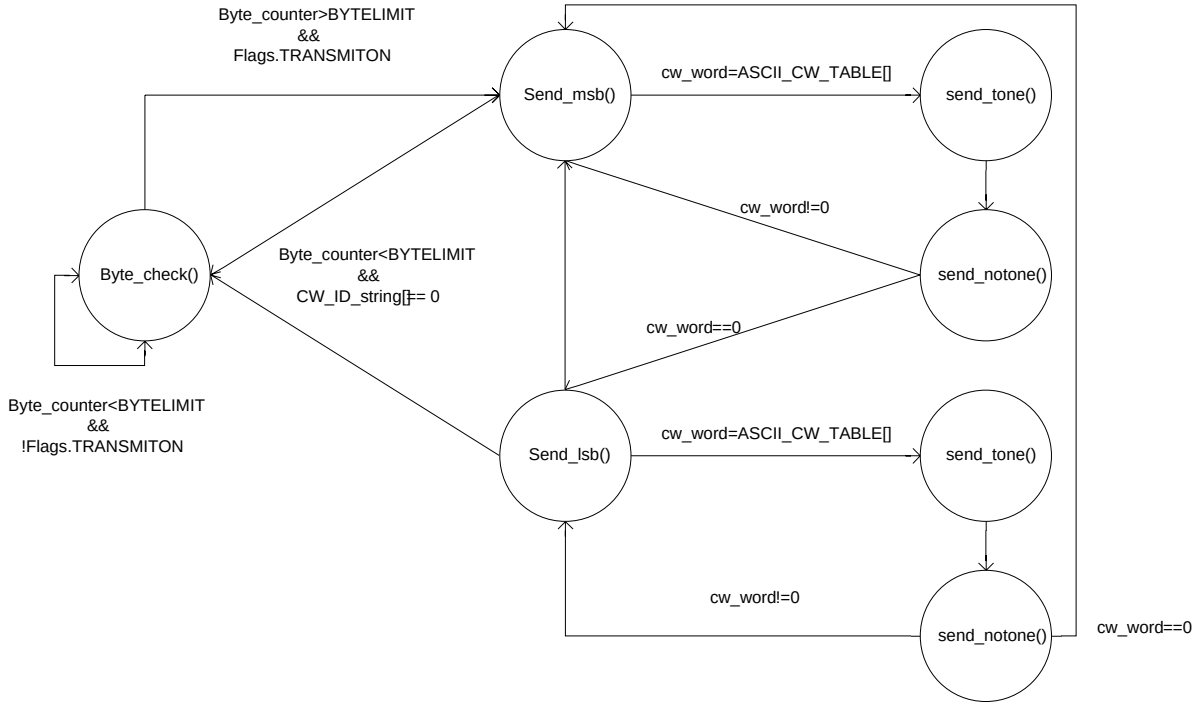
Values of 0-63 correspond to frequencies from 1500 to 2300Hz.
These values are scaled and sent to the SSTVAOUT.ASM module for outputting the proper tones to a transmitter.

Below is a diagram of the state machine for generating a CW ID after a SSTV transmission. An ASCII string containing the users call sign is stepped through character by character and a look-up table is used to get the proper sequence of dots and dashes. The table is indexed into by an ASCII value from 0 to 127 and the table returns a 16 bit value formatted in the following manner. The word in the CW table is divided into 8 groups of two bits starting at the msb side. The two bits represent one of four possible states.
00 - end of character
01 - DOT
10 - DASH
11 - SPACE of two dot times
For example, for the letter “L”(value is 76) the table entry used is:

```
.word   0110010111000000b  ;( 76)  L    .-..
```

Not all punctuation characters are implemented with this table.

CW ID State Diagram



SSTVAOUT.ASM Module

This module generates the transmit tone frequencies for SSTV and also the CW ID tones. Basically if the modem is in the transmit mode and SSTV is being sent, an NCO is used to generate tones depending on the current value of “TXfreq” which is calculated in the SSTVIN.ASM module. “TXfreq” is used as the phase increment value in an NCO oscillator after it is passed through a low pass FIR filter to roll off any abrupt frequency changes that may occur. This probably isn’t needed but there are plenty of RAM and cpu cycles available.

If the CW ID is being sent, then the value in “TXfreq” is actually the amplitude value while the frequency is constant at 700 Hz. This amplitude value is run through a simple running average routine to roll off the edges of the CW signal to prevent key clicks. A FIR filter could be used if one wanted to decimate the sample frequency down low enough to get the low cutoff frequency needed to filter the CW signal. The averaging filter works well enough in this application.

When not transmitting, the D/A is fed with the receive PLL NCO output so that SSTV can also be monitored using the usual sound card inputs or limiter circuits with the slight advantage that the DSP filters and PLL detector might give.

Below is some C code from the SSTVAOUT.ASM module comments.

```
;void Service_Aout()
;{
;  if( AR4 != AR5 ){            // if transmit Sample_Que is not full
;     Sample_Que[ AR4++ ] = OutBuf;
;     if( AR4 > Sample_Que + SAMPQUESIZE-1 )
;        AR4 = Sample_Que;
;     if( Flags.TRANSMITON ){           // if transmit mode is active
;        if( Flags.CWID_ACTIVE ){
;           CW_Sum = CW_Sum + TXfreq - CW_Ave;  //TXfreq is actually
;           CW_Ave = CW_Ave/64;                 //the amplitude modulation
;           TXphzacc = TXphzacc + CWFREQ;       // update new phase
;           Temp1 = SIN_TABLE[ TXphzacc>>8 ];   // create TX sin values
;           OutBuf = CW_Ave*Temp1;
;        }
;        }else{
;           LPFdelay[ 0 ] = TXfreq;     // do Low pass FIR calculation
;           acc = 0;
;           for( i=LPFIR_LENGTH-1; i >= 0; i-- ){
;              acc += LPFdelay[ i ] * LPFIRCOEF[ i ];
;              LPFdelay[ i+1 ] = LPFdelay[ i ];  // do the shuffle
;           }
;           Temp0 = acc;
;           TXphzacc = TXphzacc + Temp0;        // update new phase
;           OutBuf = SIN_TABLE[ TXphzacc>>8 ];  // create TX sin values
;        }
;     }else{
;        OutBuf = nco_sin;                      //if not xmitting
;        TXphzacc = 0;                          // send receiver nco
;     }
;  }
;}
;. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

SSTV MODEM TEST METHOD

Several methods were employed in debugging and verifying the modem design. The primary measurement tool was an oscilloscope. For measuring timing, digital outputs were used such as LED ports or TNC port bits. For measuring signal data, the D/A channel of the DSP-93 was used to output various test points.
A Telulex SG-100 signal generator was very useful in providing an assortment of AFSK signals for tuning the PLL and measuring the demodulator performance.
JVFAX 7.1, EZSSTV, and W95SSTV were used to generate and compare images.

SSTV MODEM PERFORMANCE

On strong SSTV signals there is no difference in performance with reception using the DSP-93 or using W95SSTV and a sound card. On weak signals in the noise, there is a narrow S/N region where there is a slight advantage using the DSP-93 of maybe a dB but rarely does a signal hover around the same S/N ratio. It performs much better than the simple op-amp limiter demodulators especially on noisy signals. Perhaps the best use of a DSP for SSTV would be in providing a “brick wall” filter function prior to the decoding using a sound card interface.

The processor load was roughly measured by measuring the peak and average time it took to service all seven software modules in the main code loop. The minimum time around the loop was 15 uSec. Peak processing time around the loop while receiving and transmitting, was around 90 uSec. This means the Sample_Que probably never gets even one sample behind. The average time around the loop was about 60 uSec. This means the processor is running about 65% of a full load at the present sample period of 91.8uSec. Code size takes about 2.1K of program space.

Areas for Improvement:

Most of the improvements could be made in the PC application side. JVFAX has problems with pixel registration(the red, green, and blue pixels don’t align correctly) especially with the Scottie 1 mode. It places the horizontal sync pulse in between the wrong color planes on receive and transmit making every line off in registration.

The other problem is that only 6 bits of resolution is allowed by the JVFAX serial mode whereas the DSP-93 has about 12 bits per color of useable resolution. This limits the color depth of images and can cause contouring in areas of the image.

JVFAX is a DOS based program and is soon going to be obsolete as GUI based operating systems take over.

Ideas for the Future:

A Windows application could be written to interface the serial port to the DSP-93 and take advantage of all 8 bits in each sample byte.

Also the DSP-93 could provide additional information for each pixel such as whether the data is valid or just noise so that the application program could intelligently fill in noisy areas of the image.

These ideas are not likely to be implemented as the W95SSTV program by Jim Barber, N7CXI and William Montgomery, VE3EC does a great job using a $50 sound card.

References

- Marvin E. Frerking, “Digital Signal Processing in Communication Systems”
- K. Sam Shanmugam, “Digital and Analog Communication Systems”
- Texas Instruments, “TMS320C2x Users Guide”
- Texas Instruments, “Digital Signal Processing Applications with the TMS320 Family Theory, Algorithms, and Implementations” Vol. 2
- Robert McGwier, N4HY, Notes from HFSIG posting.