Modems 2

2.1 OVERVIEW

The International Telegraph and Telephone Consultative Committee (CCITT), which determines protocols and standards for telephone and telegraph equipment, has authored a number of recommendations describing modem operation. This chapter surveys the fundamental algorithms of the V.32 modem recommendation, which describes the operation of a high-speed modem. Implementations of the algorithms on the ADSP-2100 family of DSP microprocessors are shown.

A modem is an electronic device that incorporates both a **mo**dulator and a **dem**odulator into a single piece of signal conversion equipment. Interfacing directly to the communication channel, modems establish communication links between various computer systems and terminal equipment. In most cases the communications channel is the general switched telephone network (GSTN) or a two- or four-wire leased circuit. The GSTN is, for the most part, a copper wire network. The bandwidth of this channel is limited to 200 Hz to 3400 Hz.

Traditionally, a modem was implemented using analog discrete components. Today, digital circuits centered around a high performance digital signal processor can meet the demands of modem algorithms without the difficulties associated with analog circuitry. A digital modem implementation offers programmability, temperature insensitivity, ease of design and often reduced cost when compared with analog implementations.

2.2 V.32 MODEM DEFINITION

The V.32 recommendation describes a full duplex synchronous modem that operates on the general switched telephone network (GSTN) as well as point-to-point leased circuits. The V.32 modem communicates at a rate of 9600 bits per second (with a 4800 bit per second slow down mode) utilizing quadrature amplitude modulation (QAM). Four-bit symbols (bauds) modulate a carrier frequency of 1800 Hz with a modulation rate of 2400 bauds per second. *The modulation of 4-bit symbols at a rate of 2400 symbols per second yields the 9600 bit per second specification.*

There are three signal coding modes to choose from in the V.32 recommendation.

- 9600 bit/second 16-point QAM. Four bits per symbol are transmitted.
- 9600 bit/second 32-point trellis-coded QAM. Transmitted symbols contain four information bits and an additional trellis encoded bit for error correction.
- 4800 bit/second 4-point QAM.

The second method, which produces a redundant bit for error correction, is the method used in the implementation described in this chapter.

Channel separation is achieved through echo cancellation. Echo cancellers are subject to CCITT specification G.165. An ADSP-2100 family implementation of an echo canceller is described in this chapter.

The V.32 modem transmits with a carrier frequency of 1800 ±1 Hz and must be able to operate with received carrier frequency offsets of ±7 Hz. The V.32 recommendation also specifies the transmitted spectrum.

2.2.1 Transmitter Algorithms

A block diagram of the transmitter section of the V.32 modem implemented in this chapter is shown in Figure 2.1. The input serial bit stream is subject to a number of algorithms prior to modulation and transmission. Each step is described briefly below and in greater detail in the following sections.

Scrambler. The input serial bit stream is first scrambled by a self-synchronizing (requires no clock signal) scrambler. Scrambling takes the input serial bit stream and produces a pseudo-random sequence. The purpose of the scrambler is to whiten the spectrum of the transmitted data. Without the scrambler, a long series of identical symbols could cause the receiver to lose carrier lock. Scrambling makes the transmitted spectrum resemble white noise, to utilize the bandwidth of the channel more efficiently, makes carrier recovery and timing synchronization easy and makes adaptive equalization and echo cancellation possible.

Encoders. The scrambled bit stream is divided into groups of four bits. The first two bits of each 4-bit group are first differentially encoded and then convolutionally encoded. This produces a 5-bit symbol in which the first bit is a redundantly coded bit.

Modems 2

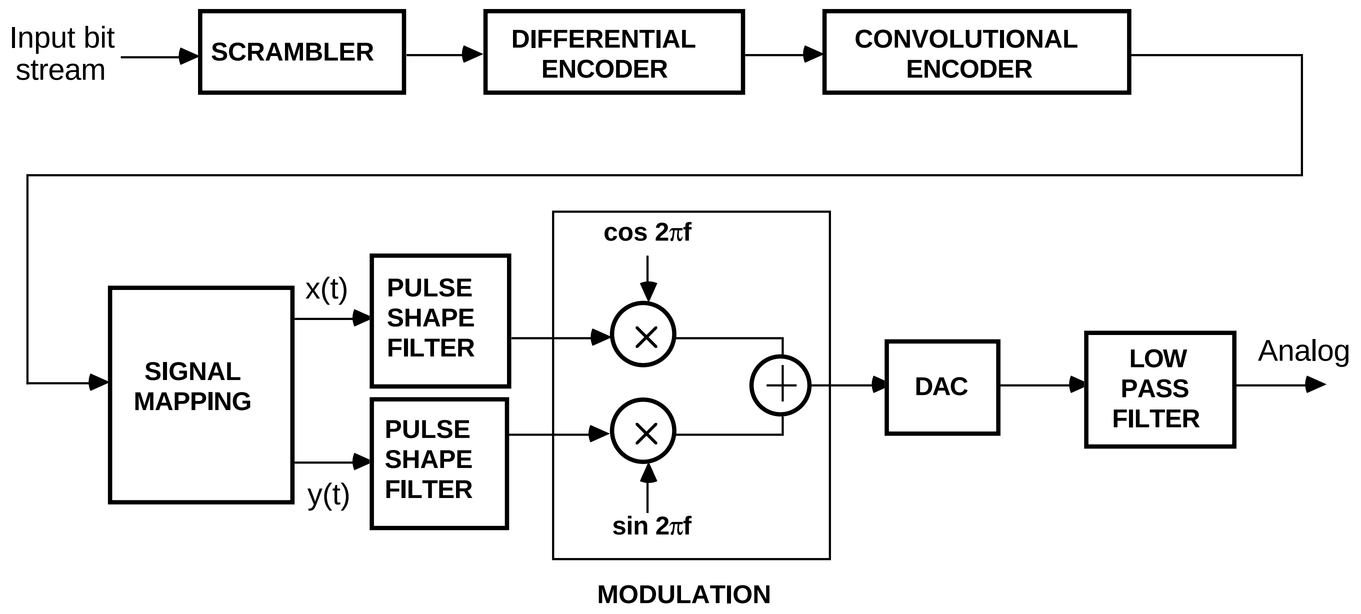

Figure 2.1 Transmitter Block Diagram

Signal Mapping. The 5-bit symbols are mapped into the signal space (defined in the V.32 recommendation) for modulation. The signal space mapping produces two coordinates, one for the real part of the QAM modulator and one for the imaginary part.

Pulse Shape Filters. The pulse shape filter is based on the impulse response of a raised cosine function. Used prior to modulation, these filters attenuate frequencies above the Nyquist frequency that are generated in the signal mapping process. The filters are designed to have zero crossings at the appropriate frequencies to cancel intersymbol interference.

Modulation. The modulation for all coding schemes in the V.32 modem recommendation is quadrature amplitude modulation (QAM). The carrier frequency is 1800 Hz and the modulation rate is 2400 symbols/second.

After modulation, the samples are converted to an analog signal. The analog output is filtered through a smoothing filter.

2.2.2 Receiver Algorithms

A block diagram of the receiver section of the V.32 modem described in this chapter is shown in Figure 2.2. Each step is described briefly below and in greater detail in the following sections.

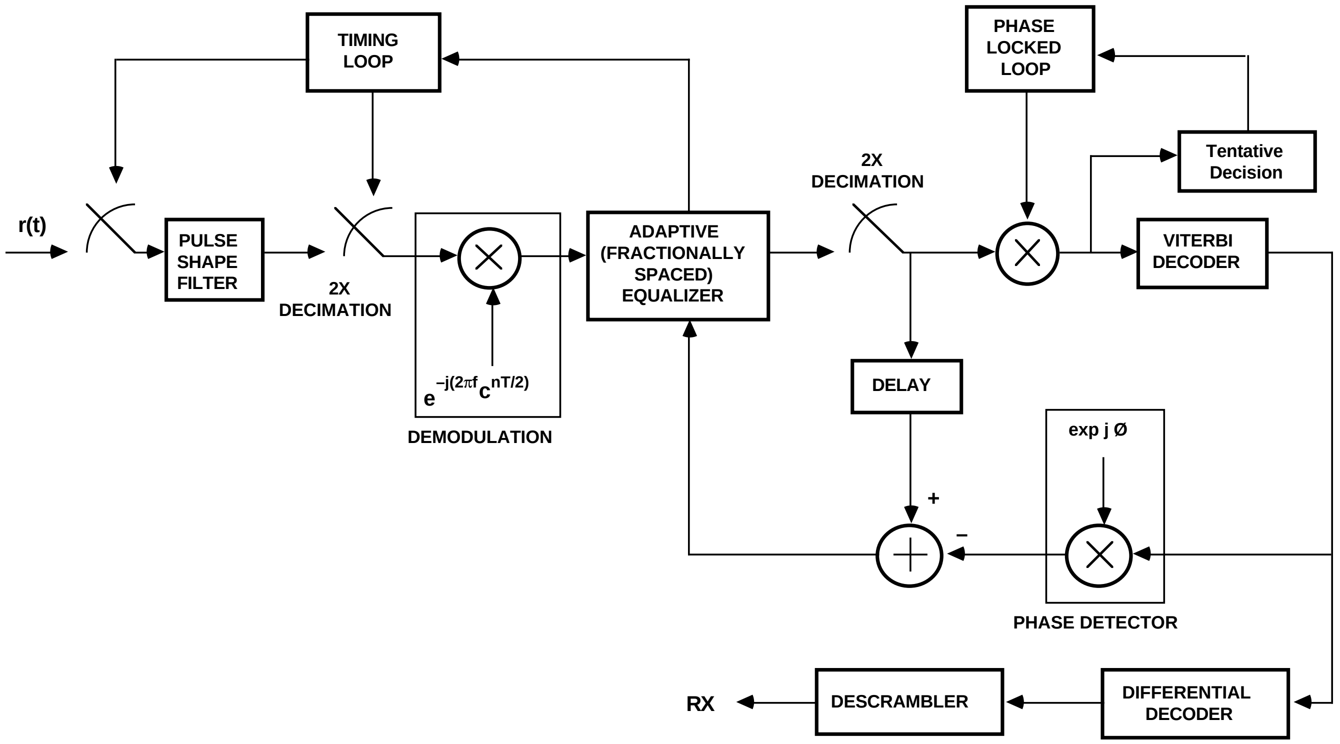

Figure 2.2 Receiver Block Diagram

Input Filter. The received analog signal is oversampled by a factor of 4 at 9600 samples per second. The sampled input is filtered with a raised cosine pulse shape filter. The output is then decimated by a factor of 2.

Demodulation. Multiplication by $e^{-j(2\pi fCnT/2)}$ demodulates the signal. QAM demodulation techniques are described in this chapter.

Adaptive Equalizer. An adaptive equalizer compensates for distortions introduced in the communications channel. A 64-tap fractionally spaced equalizer provides the performance necessary for V.32 applications. The equalizer also feeds a timing loop which adjusts the 4X sampling input and the 2X sampling output of the input filter. An ADSP-2100 family implementation of an adaptive equalizer is described in this chapter.

Viterbi Decoder. The decoder takes as input a demodulated, pulse shaped, equalized signal. The Viterbi algorithm is employed as a decoder in order to determine the appropriate signal constellation point received. This algorithm is a soft-decision maximum likelihood sequence decoder. By keeping a past history of 20 or so baud, the decoder can determine the signal point received in noisy conditions. The phase detector and delay adjust the feedback from the Viterbi decoder to the equalizer, which is constantly adapting in response to the received data.

Differential Decoder and Descrambler. Once the amplitude and phase of the signal point received is known, the corresponding symbol must be back-mapped to decode the encoded bits. The decoded 4-bit symbol is then descrambled utilizing the same generating polynomials as the scrambler.

2.2.3 Scrambler

The V.32 modem recommendation calls for the use of a scrambler in the transmit section of the modem and descrambler in the receive section of the modem. The scrambler and descrambler are based on simple polynomials. Each transmission direction uses a different scrambler, i.e., a different generating polynomial, as specified in the V.32 specification. The calling or call mode modem uses the following generating polynomial (GPC):

$$GPC = 1 + x^{-18} + x^{-23}$$

where x is the input sample and the exponent on x indicates a time delay, e. g., x^{-23} is the twenty-third previous sample. The answering or answer mode modem uses a similar scrambler with the following generating polynomial (GPA):

$$GPA = 1 + x^{-5} + x^{-23}$$

The additions are modulus 2 additions, that is, the bitwise exclusive-OR of the data values. The transmitting modem scrambles the input data sequence by dividing the message sequence by the generating polynomial. The receiving modem multiplies the scrambled sequence by the same polynomial to descramble and recover the original message sequence.

These polynomials can be thought of as digital filters. The scrambler has an all pole transfer function and the descrambler has an all zero transfer function.

The scrambler output is pseudo-random. For a repetitive input signal, the scrambler output is also repetitive with a maximum period of $2^k–1$ samples, where k is the order of the generating polynomial (23 in the case of the V.32 scrambler). In order to maximize the period of the pseudo-random output patterns, the specified GPC and GPA are irreducible and primitive.

A block diagram of the call mode scrambler is shown in Figure 2.3; x_{in} is the serial bit input stream and D_S is the scrambled data bit stream. Each delay block corresponds to a serial port cycle and each addition block is an exclusive OR operation.

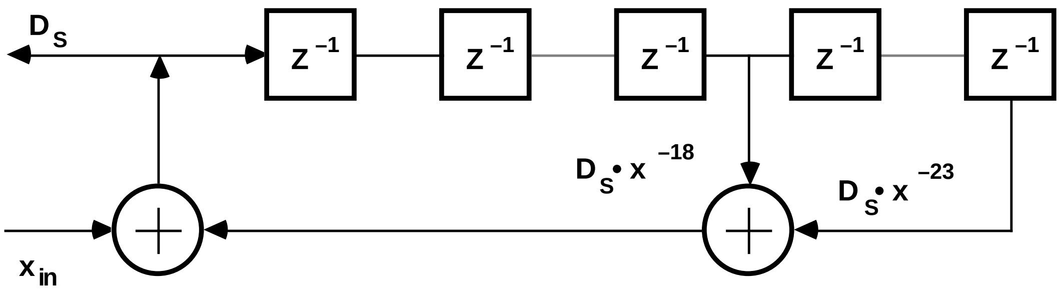

Figure 2.3 Call Mode Scrambler

The answer mode scrambler block diagram (Figure 2.4) is similar. The fifth delay line sample, x^{-5}, is used in the answer mode scrambler rather than the eighteenth delay line value as in the call mode scrambler.

2.2.4 Descrambling

The descrambler is implemented using a delay line, similar to the scrambler. The descrambler is the last functional block that the data passes through in the receiver. The data that is input to the descrambler is in effect multiplied by the appropriate generating polynomial. This multiplication performs the inverse operation of the scrambler.

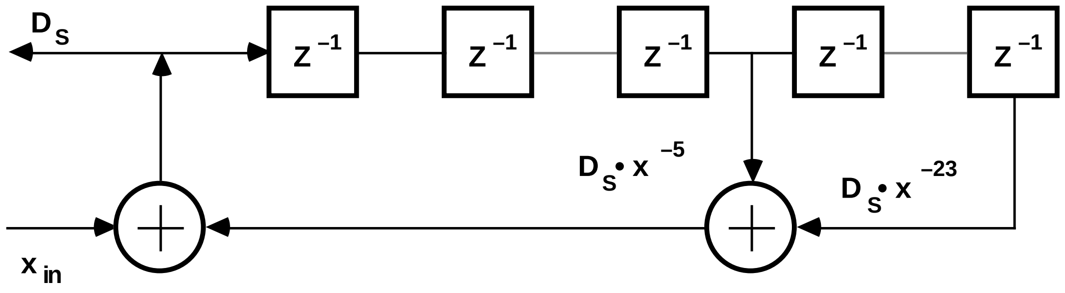

Figure 2.4 Answer Mode Scrambler

There are two versions of the descrambler, one for call mode and one for answer mode. Block diagrams for the call mode and answer mode descramblers are shown in Figures 2.5 and 2.6.

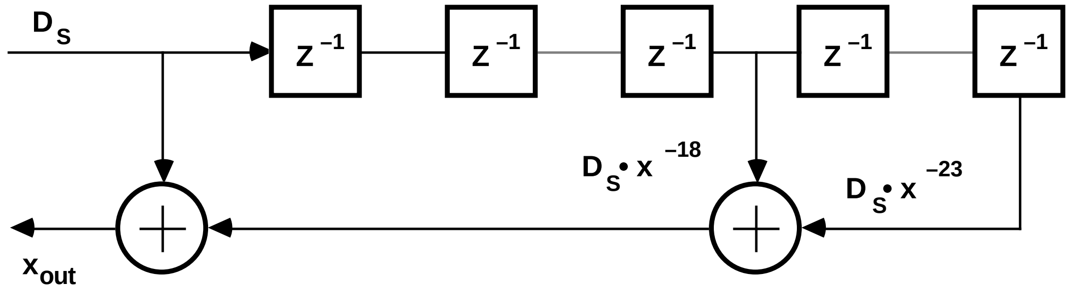

Figure 2.5 Call Mode Descrambler

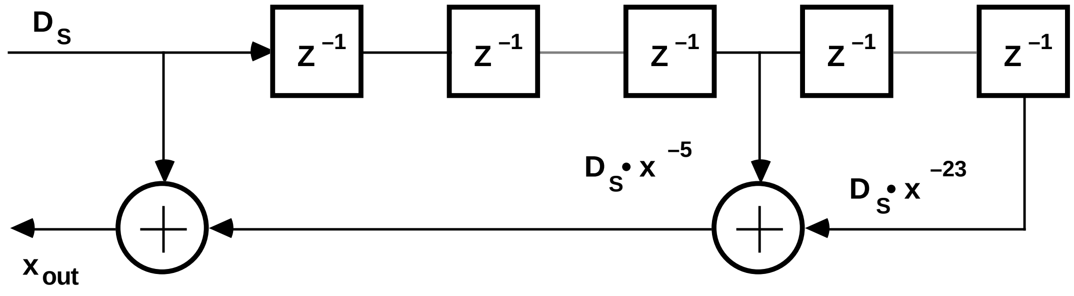

Figure 2.6 Answer Mode Descrambler

2.2.5 ADSP-2100 Family Implementation

Fundamentally, the implementation of the generating polynomials for scrambling and descrambling is the management of a delay line. The scrambler generates its output from the current input bit and two delayed outputs. The call mode uses the eighteenth and twenty-third previous outputs, while the answer mode uses the fifth and twenty-third previous outputs.

The ADSP-2100 family processors have two key features to facilitate efficient delay line management. First, each of two independent data address generators (DAGs) has four independent data pointers. An index register pointer can be programmed to handle each of the delay values and can be separately updated. Second, the DAGs support circular buffers into which delay lines are easily mapped.

In either scrambler, the twenty-third value is the oldest value, and once used is no longer needed. Thus the newest value can be written over it, so the circular buffer always contains only the 23 most recent values. Figure 2.7 illustrates the circular buffer implementation and shows the appropriate pointers.

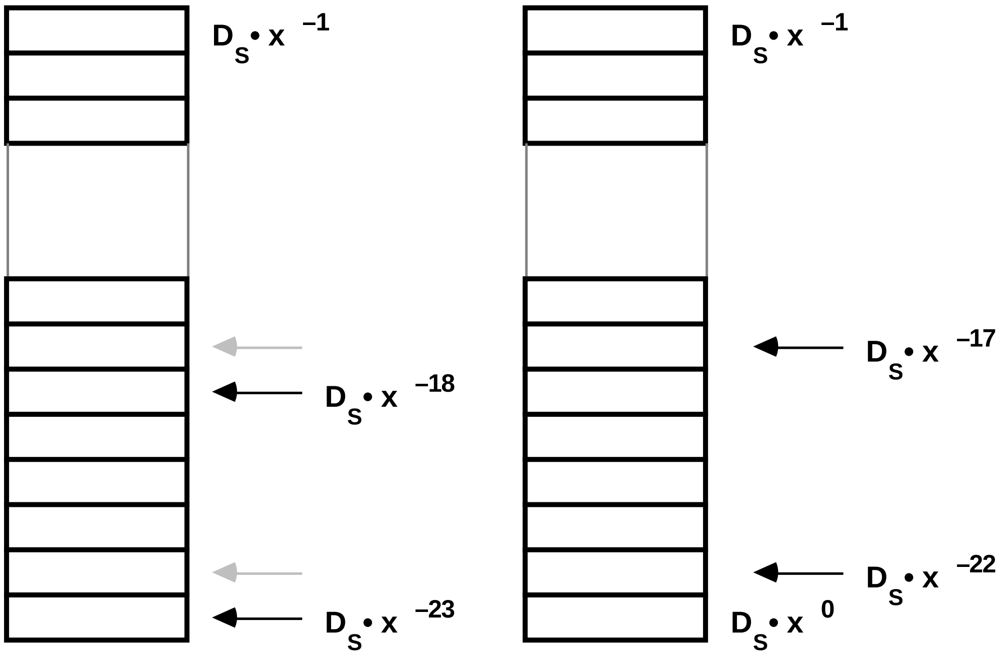

Figure 2.7 Circular Buffer Implementation For Scrambler

The value x^0 is the current input value. This value is put into an ALU register. The delayed value, $D_S \bullet x^{-18}$, is read from the circular buffer using the address supplied by a pointer (represented in the above diagram with an arrow). Once the location is read, the pointer is decremented to the next location in the buffer, shown with the light arrow. The oldest value is then written to an ALU register; the pointer's address is not yet modified. The necessary XOR operations are performed and the result is output, as well as written to the last buffer location. This pointer is now decremented to the next value, now the oldest.

This process is repeated with each new input bit. When a pointer comes to the first location in the circular buffer and is decremented, it wraps around to the last location in the circular buffer. Eighteen and twenty-three unit delays are maintained in the circular buffer, with no need to move data values, just pointer addresses.

The answer mode scrambler works similarly, except with a delay of five units instead of eighteen units. The descrambler, for both call and answer modes, also uses the same basic structure, but with a different flow of data to accomplish the inverse operation.

2.2.6 Scrambler/Descrambler Programs

The code in Listings 2.1 and 2.2 implements the V.32 scrambler (call mode) on the ADSP-2100 family processors. There are two modules, a main module and a scrambler module. The main module sets up interrupts, initializes the appropriate registers for interrupt control, initializes index registers for maintenance of the circular buffer, clears the circular buffer to zero and waits in an infinite loop for an interrupt. The only interrupt active in this program is IRQ3. This is the highest priority interrupt, and in this case it corresponds to a sampling interrupt. When a sample is ready to be scrambled, this interrupt is asserted.

The second program module is the actual scrambling routine. Included as part of this module is the *bits* subroutine, which takes 16-bit data values and strips off bits one at a time. The output of this subroutine is a string of simulated serial data values in the most significant bit position of 16-bit words. That is, a 16-bit word is input and 16 words (each of whose value is either H#8000 or H#0000) are output. These simulated serial bits are then passed to the scrambler. The scrambler output is in the AR register at the end of each pass and is written to the data memory location *dac*.

The descrambler program, in Listing 2.3, has the same fundamental structure as the scrambler program, performing the inverse operation of the scrambler.

```
.MODULE/RAM/ABS=0 cms_main_routine;

{     This module initializes registers, clears a buffer}
{     of length 23 for the call mode scrambler, sets IMASK}
{     and waits in a loop for sampling interrupt}
{     CALLS:       initial, clear_buffer}
{     INTERRUPTS: only interrupt 3 active}

.CONST              no_bits_per_word=16;
.VAR/DM/RAM/CIRC    buffer[23], input_buffer [no_bits_per_word];
.GLOBAL             input_buffer;
.PORT               cntl_port;
.EXTERNAL           start_scramble;

{interrupt jump table}
               RTI;                 {only INT3 is used}
               RTI;
               RTI;
               JUMP start_scramble; {INT3 8 kHz from codec}

{main routine}
               CALL initial;
               CALL clear_buffer;
               IMASK=H#8;           {enable interrupt 3}
mainloop:      JUMP mainloop;       {loop until interrupted}

{————————————INIT SUBROUTINE—————————————}
{One time initialization subroutine, sets up registers}

initial:       IMASK=B#0000;        {disable interrupts}
               ICNTL=H#F;           {edge sensitive interrupts}
               SI=0;
               DM(cntl_port)=SI;    {load codec control register}

               L0=%buffer;          {length registers}
               L1=%buffer;          {circular buffer length 23}
               L2=%buffer;
               L3=0;                {no other index circ buffer}
               L4=0;
               L5=0;
               L6=0;
               L7=0;
                                    {index registers}
               I0=^buffer;          {ds(n-5)}
               I1=^buffer + 17;     {ds(n-18)}
               I2=^buffer + 22;     {ds(n-23)}
```

```
                I3=0000;
                I4=^input_buffer + 15;

                M0=0;                     {modify registers}
                M1=-1;
                M2=1;
                M4=-1;
                M5=1;
                SE=4;                     {SE for nibble pack}
                RTS;

{——————————CLEAR BUFFER SUBROUTINE———————}
{initialize scramble buffer to zero}

clear_buffer:  CNTR=%buffer;
               DO clear UNTIL CE;
clear:            DM(I0,M1)=0;
               RTS;

.ENDMOD;
```

Listing 2.1 Call Mode Scrambler Main Routine

```
.MODULE         call_mode_scrambler;

{     This module performs V.32 call mode scrambling}
{     The generating polynomial is: xin+y(n-18)+y(n-23)}
{     CALLS:  bits}

.EXTERNAL       input_buffer;
.CONST          no_bits_per_word=16;
.PORT           codec;
.PORT           dac;
.ENTRY          start_scramble;

start_scramble:   AY0=DM(codec);    {read from port}
                  CALL bits;        {show as serial stream}
                  CNTR=no_bits_per_word;  {scramble 16 times}
                                    {once for every bit of input}
                  DO scrambl UNTIL CE;
                     AY0=DM(I4,M5);
                     AX0=DM(I1,M1);  {d(n-18)}
                     AY1=DM(I2,M0);  {d(n-23)}
                     AR=AX0 XOR AY1; {d(n-18) + d(n-23)}
                     AR=AR XOR AY0;  {d(n) + d(n-18) + d(n-23)}
                     DM(I2,M1)=AR;   {store scramble in buffer}
                                     {write new value over oldest}
                     DM(dac)=AR;     {out to dac}
                     MODIFY(I4,M4); {reset pointer to last buffer}
                                    {value for next input word}
scrambl:             NOP;

                  RTI;


{——————————BITS SUBROUTINE—————————————}
{ takes output from u_expand (16-bit word) and separates out }
{ the bits; stores as MSB in a 16-word buffer ‘input_buffer’}
{ The most significant bit of the input word is at the top of }
{ the buffer}

bits:             AX0=AY0;              {expanded output into ALU}
                  SE=15;
                  CNTR=no_bits_per_word;
                  AY0=H#8000;
                  DO bit_loop UNTIL CE;
                     AR=AX0;
                     SR=LSHIFT AR (LO);  {shift so next bit is}
                                         {MSB in reg SR0}
```

```
                      AR=AR0 AND AY0;  {mask out all except MS}
                      DM(I4,M4)=AR;
                      AY1=SE;          {decrement SE for next}
                      AR=AY1-1;
bit_loop:             SE=AR;
                   I4=^input_buffer;
                   SE=4;
                   RTS;

.ENDMOD;
```

Listing 2.2 Call Mode Scrambler Scrambling Routine

```
.MODULE/RAM/ABS=0 main_routine;

{     Descrambling Routine }
{     Call Mode Functions implemented:}
{           d(n)=di(n)+d(n-18)+d(n-23)}

{     System file:      fullpm.sys}
{     CALLS:      initial, clear_buffer, output}

.VAR/DM/RAM/CIRC  buffer[23];
.PORT             codec;
.PORT             dac;
.PORT             cntl_port;

                  RTI; RTI; RTI;          {int0-2 not used}
                  JUMP start_descramble;  {INT3 8 kHz from codec}
                  CALL initial;
                  CALL clear_buffer;

                  IMASK=h#8;        {enable interrupts}
mainloop:         JUMP mainloop;    {loop until interrupted}

{———————— descramble subroutine ————————————}
{addressing circular buffer with 2 pointers for modem scrambler}

start_descramble: AY0=DM(codec);    {read from port}
                  AX0=DM(I1,M1);    {d(n-18)}
                  AY1=DM(I2,M0);    {d(n-23)}
                  AR=AX0 XOR AY1;   {d(n-18)+d(n-23)}
                  AR=AR XOR AY0;    {d(n)+d(n-18)+d(n-23)}
                  DM(I2,M1)=AY0;    {store scramble in buffer}
                                    {input stored... not output}
                  CALL output;
                  AR=0;             {clear AR for next time}
                  RTI;


{—————————— initialize subroutine ——————————}
{initialize registers}

initial:          IMASK=B#0000;     {disable interrupts}
                  ICNTL=H#F;        {edge level interrupts}
                  SI=0;
                  DM(cntl_port)=SI; {load codec control reg}
                  L0=%buffer;       {circular buffer length 23}
                  L1=%buffer;
                  L2=%buffer;
                  L3=0;
                  L4=0;
                  L5=0;
                  L6=0;
                  L7=0;
                  I0=^buffer;
                  I1=^buffer + 17;
                  I2=^buffer + 22;
                  M0=0;
                  M1=-1;
                  SR0=0;
                  SR1=0;
                  SE=16;
                  RTS;

{——————— clear buffer subroutine ———————————}
{initialize buffer to zero}

clear_buffer:     CNTR=%buffer;
                  DO clear UNTIL CE;
clear:               DM(I0,M1)=0;
                  RTS;

{———— output routine packs serial into 16 bit words ————}
output:           SR=SR OR LSHIFT AR(LO);
                  AY0=SE;
                  AR=AY0 -1;
                  SE=AR;
                  IF EQ CALL out;
                  RTS;

out:              DM(dac)=SR1;
                  SR0=0;
                  SR1=0;
                  SE=16;
                  RTS;

.ENDMOD;
```

Listing 2.3 Call Mode Descrambler Routine

2.2.7 Raised Cosine Filter

For the V.32 modem recommendation, 5-bit symbols are modulated by a carrier of 1800 Hz. This modulation is performed digitally. Coupled with the modulator and the demodulator are pulse shaping low pass filters. These digital filters eliminate intersymbol interference (ISI) on the bandlimited GSTN.

A brief development of the theory of pulse shaping filters follows. For a more complete theoretical discussion of pulse shaping filters, see “References” at the end of this chapter: Bingham, Lee and Messerschmitt, Proakis.

Low pass transmitted signals can be shown to have the form

$$\sum_{n=0}^{\infty} I_n \, g(t-nT)$$

where I_n is the discrete code word and g(t) is a pulse. For the bandlimited channel, we desire a transmitted pulse g(t) that produces no ISI. If the channel is ideally bandlimited, then an ideally bandlimited pulse can be used. In the frequency domain, this ideally bandlimited pulse can be described as:

$$G(f) = \begin{cases} T & \text{for } f < 1/2T \\ 0 & \text{for } f \geq 1/2T \end{cases}$$

This spectrum has an ideal rectangular shape.

In the time domain, this ideal spectrum shape is the sinc function:

$$g(t) = \sin(\pi t/T)/(\pi t/T)$$

The nulls (zero values of the pulse function) occur at multiples of T, the baud rate. Because of the placement of the nulls, there is no additive interference due to previous symbols; there is no ISI.

The ideal pulse shaping filter is not practical to implement. The ideally bandlimited frequency response has a corresponding infinite impulse response. Although the impulse response has a zero value at all multiples of T, any mistiming in the modem produces an infinite series of ISI terms.

A pulse shaping filter that is practical and widely used in digital communications is the raised cosine pulse shaping filter. The raised cosine pulse shaping filter is realizable, unlike the ideal pulse shaping filter. The raised cosine function has tails that decay proportional to $1/t^3$, whereas the ideal pulse tails off proportional to $1/t$. Mistiming errors in sampling in the modem therefore have a much less dramatic effect on the amount of ISI in the raised cosine pulse filter.

A generic formula for the impulse response of the raised cosine filter, p(t), is shown below. *T* is the symbol rate in Hz, *t* is the sampling rate in Hz, and α is the rolloff factor.

$$p(t) = \frac{\sin(\pi t/T) \bullet \cos(\alpha\pi t/T)}{(\pi t/T) \bullet (1-(2\alpha\pi t/T)^{2)}}$$

The rolloff factor, α, represents the amount of excess bandwidth required. A raised cosine with a rolloff factor of 0 needs the least excess bandwidth. As α varies from 0 to 1, the amount of excess bandwidth required increases from 0 to 100%. For purposes of this implementation, a common rolloff factor of 0.25 is used. For the V.32 modem, the symbol rate, T, is specified at 2400 symbols per second. The sampling rate, t, is usually 9600 Hz. The frequency response of the raised cosine pulse shaping filter with these parameter values is shown in Figure 2.8.

The pulse shaping filter usually spans four baud intervals. For a sampling rate of 9600 Hz and a symbol rate of 2400 Hz, a 17-tap FIR filter can be used.

2.2.8 ADSP-2100 Family Implementation

The raised cosine pulse shaping filter can be implemented in the modem as a simple FIR filter. Implementation of FIR filters on the ADSP-2100 family is straightforward. The dual DAGs with circular buffering and the on-chip Harvard architecture allows for efficient realization of FIR filter structures. A complete description of FIR filters as well as other fixed-coefficient filters can be found in *Digital Signal Processing Applications Using the ADSP-2100 Family*, Chapter 5 (see “Literature” at the beginning of this book).

Filter coefficients are arrived at using the formula above, generated with a C program. The coefficients are scaled to provide a filter with 0 dB gain.

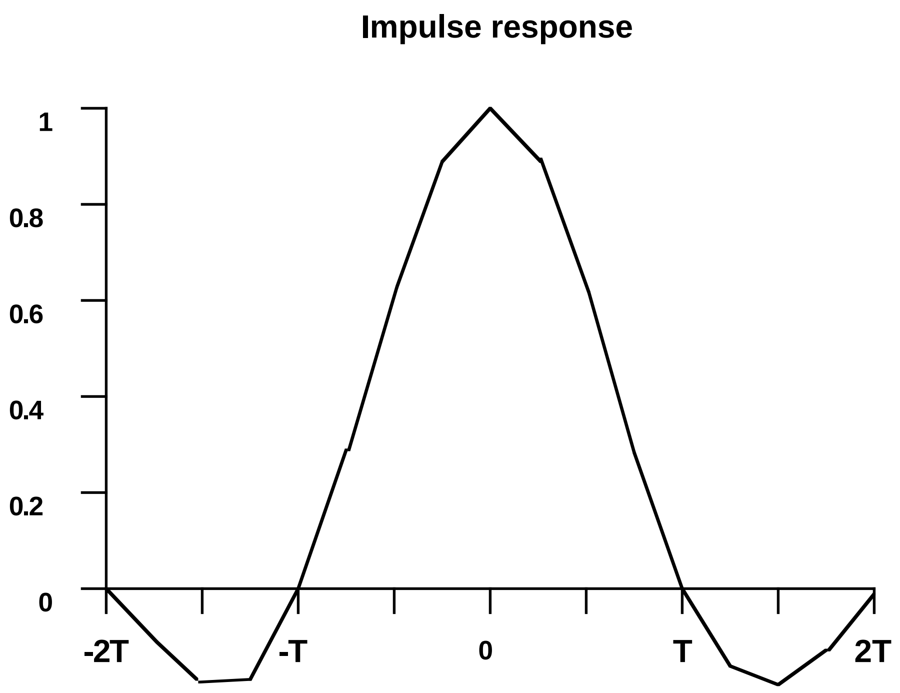

Figure 2.8 Raised Cosine Pulse Shaping Filter, α=0.25

The coefficients represent a rolloff factor of 0.25, and the generated impulse response spans four baud intervals.

For the V.32 modem, the filter input is a digitally modulated value (1800 Hz carrier). Samples are processed at the baud rate (2400 baud) and are interpolated, zero-filled, to provide filter input at a rate of 9600 Hz. Samples are processed in quadrature. Figure 2.9 shows the relationship of the filter to the digital modulator and the data rates.

Listing 2.4 contains the ADSP-2100 family code for implementation of the raised cosine filter. The coefficients can be found in the data file *coef.dat*.

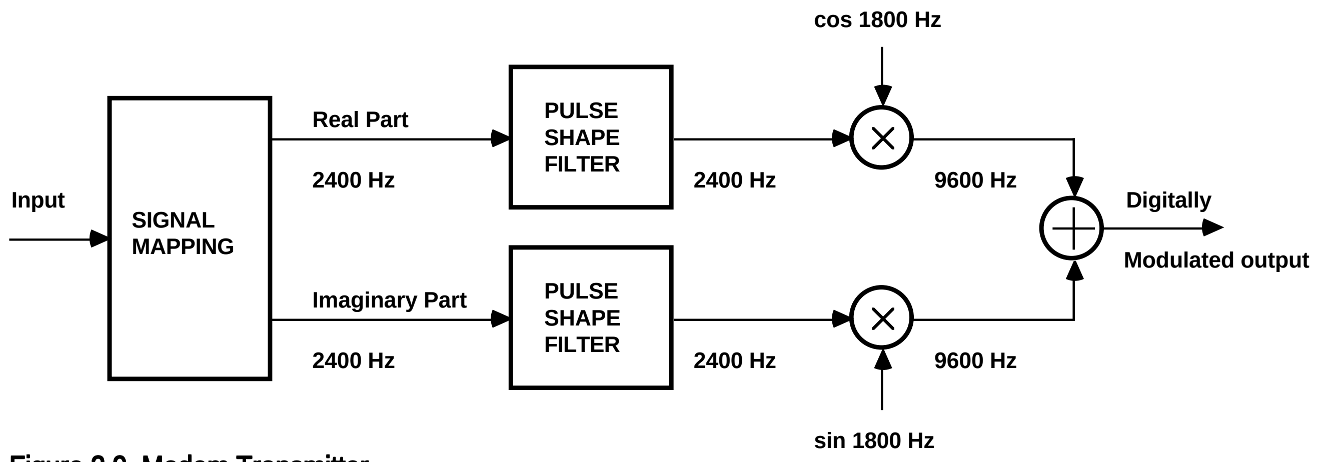

Figure 2.9 Modem Transmitter

```
.MODULE/boot=0       fir_sub;
{————————————————————————————————
        Pulse Shape filter routine for V.32
        ICASSP DEMO

        Rev History     2/8/90 take APP VOL I FIR routine
                        adapt for V.32

}

.ENTRY             pulse_shape;

.CONST             PSF_length=89;
.EXTERNAL          Real_PSF_delay_line, Imag_PSF_delay_line, Pulse_Shape_Coeff;
.EXTERNAL          real_PSF_i0, imag_PSF_i0;

.VAR/DM            psf_save_I0;
.VAR/DM            psf_save_L0;
.VAR/DM            psf_save_I4;
.VAR/DM            psf_save_L4;
.VAR/DM            test_psf1;
.VAR/DM            test_psf2;
```

(listing continues on next page)

```
pulse_shape:      DM(psf_save_I0)=I0; DM(psf_save_L0)=L0;  {save I0,L0,I4,L4}
                  DM(psf_save_I4)=I4; DM(psf_save_L4)=L4;

                  I0=DM(real_PSF_i0);
                  I4=^Pulse_Shape_Coeff;
                  L0=psf_length; L4=psf_length;

{——— Do real part of the filter.  ax0 contains the x value
        from the signal map module.}

                  DM(I0,M2)=AX0;        {dump new vals into delay line}
                  CNTR=PSF_Length-1;
                  MR=0, MX0=DM(I0,M2), MY0=PM(I4,M5);
sop:              MR=MR+MX0*MY0(SS), MX0=DM(I0,M2), MY0=PM(I4,M5);
                  IF NOT CE JUMP sop;
                  MR=MR+MX0*MY0(RND);
                  IF MV SAT MR;
                  AX0=MR1;              {filtered X in ax0}
                  DM(real_PSF_i0)=I0;

{——— Do the imaginary part of the Pulse Shape filter.  ax1 contains
        the imaginary part of the point from the signal map module. }

                  I0=DM(imag_PSF_i0);
                  DM(I0,M2)=AX1;        {dump new vals into delay line}
                  CNTR=PSF_Length-1;
                  MR=0, MX0=DM(I0,M2), MY0=PM(I4,M5);
imag_sop:         MR=MR+MX0*MY0(SS), MX0=DM(I0,M2), MY0=PM(I4,M5);
                  IF NOT CE JUMP imag_sop;
                  MR=MR+MX0*MY0(RND);
                  IF MV SAT MR;
                  AX1=MR1;              {filtered Y in ax1}
                  DM(imag_PSF_i0) = I0;

                  I0=DM(psf_save_I0); L0=DM(psf_save_L0);
                  I4=DM(psf_save_I4); L4=DM(psf_save_L4);

         RTS;
```

Listing 2.4 Raised Cosine Filter

2.2.9 Trellis Encoding

The GSTN was intended for voiceband transmission and is bandlimited 200 Hz to 3400 Hz. Data rates in excess of the upper band limit can be realized only by the transmission of multiple bits per symbol interval. Data rates of 9.6 Kbits per second can be achieved on unconditioned circuits and data rates of up to 16.8 Kbits per second can be realized on conditioned leased lines using the technique known as trellis coded modulation (TCM).

The V.32 modem recommendation specifies trellis encoding as an option. Four-bit symbols are encoded into 5-bit symbols that are made up of four information bits and a redundant bit. These 5-bit symbols are used with a 32 carrier state QAM modulator. A 2400 baud rate is used and 9600 information bits per second are transmitted. A trellis encoded scheme offers much better performance than a non-encoded scheme. It results in a much higher immunity to noise for a given error rate and can reduce the block error rate by three orders of magnitude for a given signal-to-noise ratio.

There are two fundamental types of codes used in channel encoding. Linear block codes include Hamming codes, BCH (Bose-Chadhuri-Hocquenghem) codes, Reed-Solomon codes, Galay codes and many others. The convolutional code, which is specified for V.32 modems can be implemented using a shift register and can be described using a diagram called a trellis diagram.

Suppose we can achieve a certain P_e (probability of error) in an uncoded system operating on a bandlimited channel. We can attempt to improve system performance by coding. If we add a single redundant bit to a binary symbol with *k* bits, we increase the number of waveforms that the modulator must produce from 2^k to 2^{k+1}. An increase in alphabet size on the same bandwidth requires a 3 dB increase in the signal to noise ratio to achieve the same P_e. That is, coding alone decreases the performance of the system.

Trellis coded modulation employs signal set partitioning in addition to redundant coding in order to increase the system performance. In the case of the V.32 modem, there are 32 modulator states. Of the four input bits to the encoder, only two are encoded. Two bits pass through uncoded and two bits are encoded into three output bits. The three bits provide a mechanism for dividing the 32 modulator states into 8 subsets of 4 modulator carrier states. The coded bits identify the subset of the 32 modulator states and the uncoded bits select a point within the subset. Figure 2.10 shows the input and output bits of the trellis encoder. Bits Q1 through Q4 are the input bits. Bits Q3 and Q4 pass through the encoder unchanged. Bits Q1 and Q2 are encoded to give Y1, Y2 and the redundant error correcting bit Y0. Bits Y0, Y1, Y2 identify the subset while the bits Q3 and Q4 identify the point within the subset.

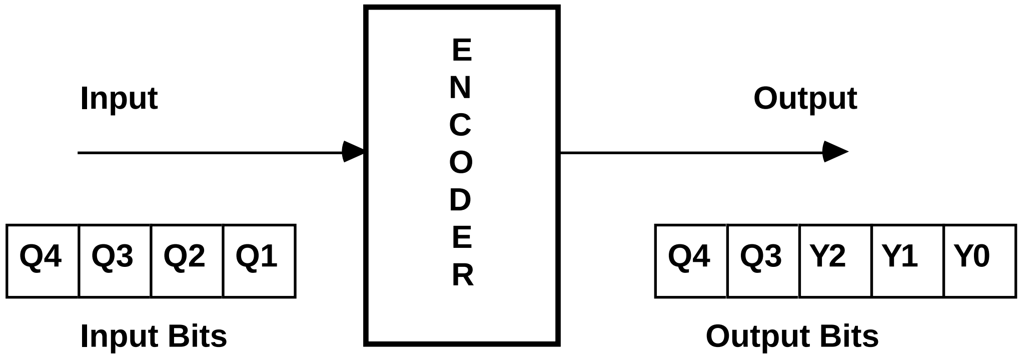

Figure 2.10 Encoder Block Diagram

The signal set for the V.32 modem (and other TCM schemes) has been designed so that there is a large distance between the members of each subset. The 32-state signal constellation for the V.32 modem is shown in Figure 2.11. Bits are ordered on this diagram left to right, most significant to least significant: Y0 Y1 Y2 Q3 Q4. The signal space mapping for the redundant coding is from Figure 3/V.32 of the V.32 recommendation.

The signal set is located on a quadratic grid known as a Z_2 lattice and the signal set type is known as 32 CROSS. In order to transmit *m* bits per signalling interval, 2^{m+1} signals are needed. The coding gain (performance of the coded signals versus uncoded signals) is approximately 4 dB for any *m*. The closest distance between any two points on the signal set is Δ_0. The closest distance between any two points in a subset (i.e., points that have the same Y0, Y1 and Y2 bits) is $\sqrt{8}\,\Delta_0$ for the 32 CROSS signal set.

All bit patterns that begin with the same three bits are spread out on the signal constellation. This signal set partitioning along with the redundant coding are the fundamentals of TCM.

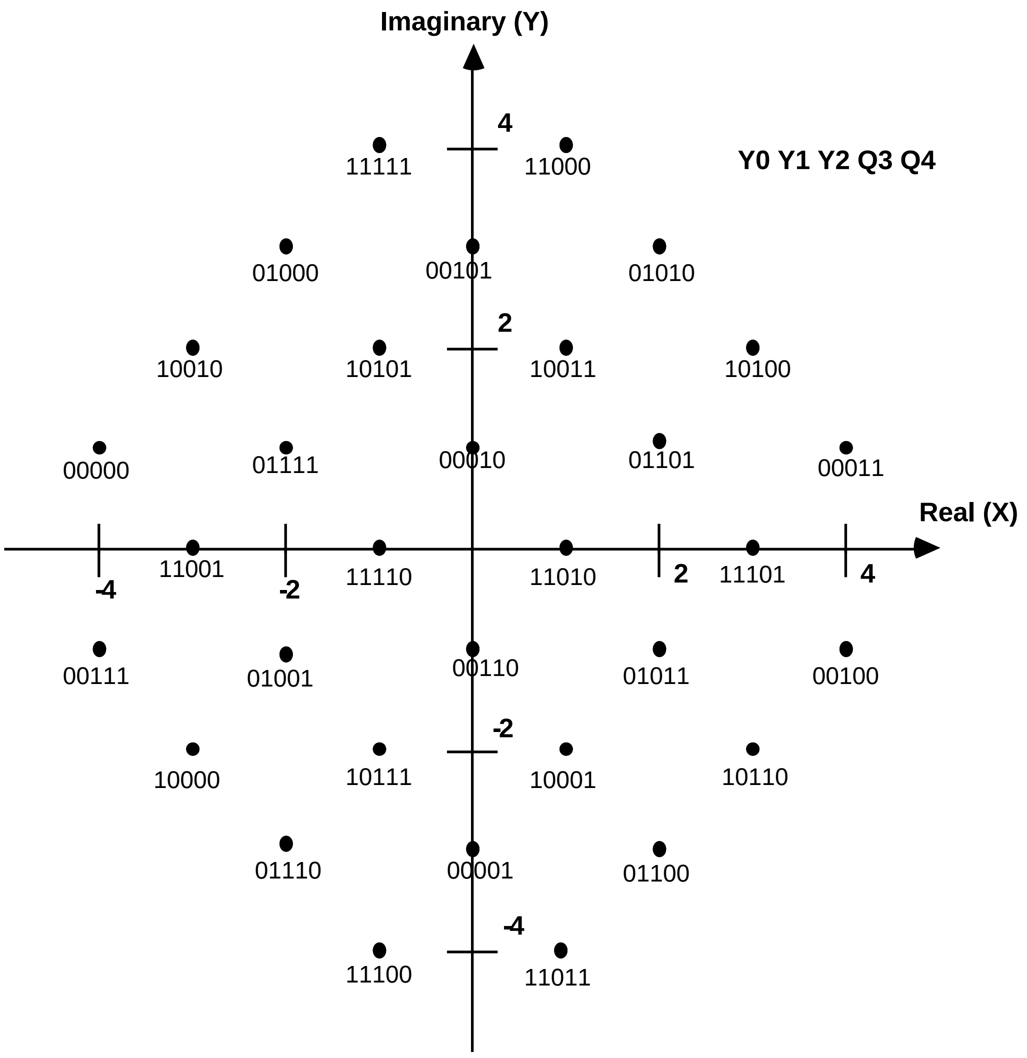

Figure 2.11 V.32 Signal Constellation

2.2.10 ADSP-2100 Family Implementation

Trellis encoding for the V.32 modem consists of two encoding operations: a differential encoder, implemented as a lookup table and a convolutional encoder, performed using a shift register and Boolean logic. Together, these two encoders generate a 5-bit symbol from a 4-bit input word.

The serial input bits to the encoder are Q1, Q2, Q3 and Q4 (Q1 first, Q4 last). Three of the output bits are Y0, Y1 and Y2, and the other two output bits are Q3 and Q4, unchanged from the input. Y1 and Y2 are generated in the differential encoder. Y0, the redundant bit for error correction, is generated in the convolutional encoder.

The differential encoder takes as input the first two bits, Q1 and Q2, and produces two output bits, Y1 and Y2. Previous output bits, Y1(n–1) and Y2(n–1) are also used in the differential encoder. The encoder is easily implemented on the ADSP-2100 family as a lookup table. The input bits and the previous output bits are combined to a 4-bit value that serves as a pointer into the lookup table. For example, assume that the current input bits are Q1=1, Q2=0, Y1(n–1)=0 and Y2(n–1)=1, for a 4-bit value of 1001. This corresponds to the 1001 (ninth) entry in the lookup table, from which the current Y1 and Y2 outputs are read. Table 2.1 shows the lookup table for differential encoding.

Inputs		**Previous Outputs**		**Outputs**	
Q1	**Q2**	**Y1(n-1)**	**Y2(n-1)**	**Y1**	**Y2**
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	1

Table 2.1 Differential Encoder Lookup Table

The convolutional encoder (Figure 2.12) uses a shift register structure to examine the four incoming bits (the output of the differential encoder) and build a 5-bit symbol. The five output bits of the convolutional encoder consist of the four input bits plus an additional redundantly coded fifth bit. This additional bit increases the complexity of the signal set, but limits the number of possible transitions between bit patterns. For any given 5-bit convolutionally encoded word, only half of the signal states can follow. In other words, the process of convolutional encoding prohibits transitions from any particular signal state to only half of the possibilities. This property is exploited in the Viterbi decoder in the receiver.

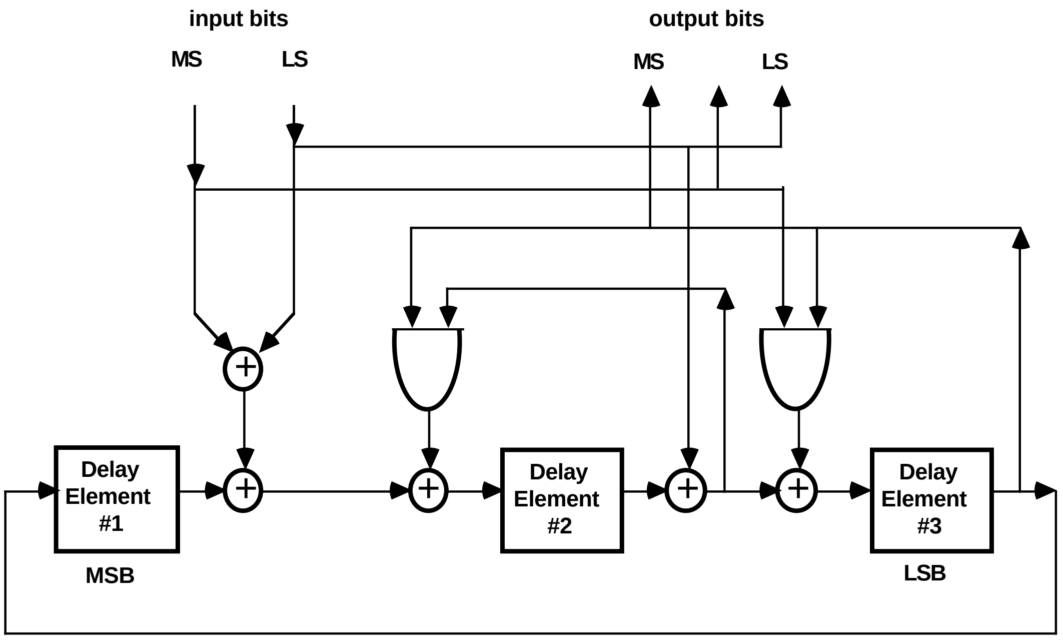

Figure 2.12 Convolutional Encoder Block Diagram

Listing 2.5 contains a ADSP-2100 family subroutine that provides both the differential encoder and the convolutional encoder. The input is assumed to be a single bit residing in the most significant bit position of a 16-bit word. Listing 2.6 shows the convolutional encoder routine that is called by the program in Listing 2.5, and Listing 2.7 contains the routine that performs signal mapping on the encoded data.

```
.MODULE/RAM      trellis;

.VAR/DM/RAM      t_table[16];
.VAR/DM/RAM      last_ys;
.VAR/DM/RAM      bit_count;
.VAR/DM/RAM      diff_out;
.VAR/DM/RAM      delay_val_1;
.VAR/DM/RAM      delay_val_2;
.VAR/DM/RAM      delay_val_3;
.VAR/DM/RAM      Y1;
.VAR/DM/RAM      Y2;
.INIT            t_table: 0,1,2,3,1,0,3,2,2,3,1,0,3,2,0,1;
.ENTRY           trellis_encode;
.PORT            dac;
.PORT            adc;
.GLOBAL          t_table, bit_count, last_ys;


{——bit count is intially 4——}
trellis_encode:  SE=DM(bit_count);
                 SI=DM(adc);        {take in new 8000 or 0000}


Q1Q2_pack:       SR=SR OR LSHIFT SI (LO); {count up 4 bits,}
                 AY0=SE;            {shift into SR register}
                 AR=AY0 -1;
                 SE=AR;
                 DM(bit_count)=SE;  {store decremented count}
                 IF EQ JUMP packed;
                 RTI;

packed:          AX0=SR1;           {stored as 4 bits}
                 AX1=4;             {Q1 Q2 Q3 Q4}
                 DM(bit_count)=AX1;
                 SR0=0;
                 SR1=0;
                 CALL d_encode;
                 RTI;
```

```
{——————————————ENCODE——————————————}
{input:  AX0 -> 0 0 0 X where X -> bits 0 0 0 0 Q1Q2Q3Q4}

d_encode:        I3=^t_table;
                 AY0=h#000C;         {mask to keep Q1 Q2}
                 AR=AX0 AND AY0;
                 AY1=DM(last_ys);    {last output Y1 Y2}
                 AR=AR XOR AY1;      {AR is Q1 Q2 Y1 Y2}

                 M3=AR;              {address in lookup}
                 MODIFY(I3,M3);      {for new Y1 Y2}

                 SI=DM(I3,M0);
                 DM(last_ys)=SI;     {AY0 ->encoded Y1 Y2}

                 AY1=3;
                 AF=AX0 AND AY1;     {keep Q3 Q4}
                 SR=LSHIFT SI BY 2(LO);
                 AR=SR0+AF;          {AR ->Y1 Y2 Q3 Q4}
                 DM(diff_out)=AR;    {store output of diff encode}
                 DM(dac)=AR;
                 CALL c_encode;      {call convolutional encode}
                 RTS;

.ENDMOD;
```

Listing 2.5 Trellis Encoder Program

```
.MODULE/RAM       conv_encode;

{  Trellis Encoder for V.32 Modem
   Implements convolutional encoder

Input:   Four bit symbols, output of the differential encoder

Output:  Five bit symbol in the LSB positions}

.VAR/DM/RAM       diff_out;         {differential encode output}
.VAR/DM/RAM       conv_out;         {convolutional encode output}
.VAR/DM/RAM       packed_4_bits;    {Q1Q2Q3Q4 as 4 LSBs}
.VAR/DM/RAM       delay_val_1;      {conv. enc delay element}
.VAR/DM/RAM       delay_val_2;      {conv. enc delay element}
.VAR/DM/RAM       delay_val_3;      {conv. enc delay element}
.VAR/DM/RAM       intermed_1;
.VAR/DM/RAM       intermed_2;
.VAR/DM/RAM       Y0;               {output bit Y0}
.VAR/DM/RAM       Y1;               {output bit Y1}
.VAR/DM/RAM       Y2;               {output bit Y2}

.GLOBAL           conv_out;
.GLOBAL           delay_val_1, delay_val_2, delay_val_3;
.GLOBAL           intermed_1, intermed_2, packed_4_bits;

.ENTRY            c_encode;
.EXTERNAL         sig_map, dac;

{——————————— CONVOLUTIONAL ENCODE———————————}
{Input is Y1Y2Q3Q4 located in “diff_out” 4 LSBs}
{Output is 3 encoded bits in data mem locations Y0 Y1 Y2}
{calls “pack_up_5_bits” for output to dac}

c_encode:   SR0=0;                  {clear shift result}
            SR1=0;
            SI=DM(diff_out);        {get input from diff encoder}
            SE=-3;
            SR=LSHIFT SI BY -3(HI); {put Y1 in LSB position}
            AY0=1;
            AR=SR1 AND AY0;         {separate Y1}
            DM(Y1)=AR;
            AX0=AR;
            SR=LSHIFT SI BY -2(HI);
            AR=SR1 AND AY0;         {separate Y2 and store}
            DM(Y2)=AR;
            AY0=AR;
            AR=AX0 XOR AY0;         {op #1}
            AY1=DM(delay_val_3);
            AR=AR XOR AY1;          {op #2}
            DM(intermed_1)=AR;

            AX0=DM(delay_val_1);
            AR=AX0 XOR AY0;         {delay val 1 XOR Y2 op #5}
            DM(intermed_2)=AR;

            AY0=DM(delay_val_2);
            DM(delay_val_3)=AY0;    {update delay val 3}
            AR=AR AND AY0;          {and_1}

            AY1=DM(intermed_1);
            AR=AR XOR AY1;
            DM(delay_val_1)=AR;     {update delay_val_1}

            AX1=DM(Y1);
            AR=AX1 AND AY0;         {and_2}
            AY0=DM(intermed_2);
            AR=AR XOR AY0;

            DM(delay_val_2)=AR;     {update delay val 2}
            DM(Y0)=AR;

            CALL pack_up_5_bits;
            RTS;

{———————————— OUTPUT FORMATTER ——————————}
{Packs up convolutional bits as 5 LSBs Y0 Y1 Y2 Q3 Q4}
{Outputs to DAC}

pack_up_5_bits:   SR0=0;            {pack up bits as Y0Y1Y2Q3Q4}
                  SR1=0;            {clear SR}

                  SR1=DM(diff_out);
                  SI=DM(Y0);

                  SR=SR OR LSHIFT SI BY 4 (HI);
                  DM(conv_out)=SR1;
                  DM(dac)=SR1;

                  SR0=0;
                  SR1=0;

                  CALL sig_map;
                  RTS;

.ENDMOD;
```

Listing 2.6 Convolutional Encoder Routine

```
.MODULE     signal_map;

{  This module takes the output of the convolutional encoder,
  that is, a five bit code residing in the LSBs of the data
  memory location “conv_out”, and looks up the x and y
coordinates
  as defined by the CCITT spec for the V.32 modem.

  The coordinates are given in the CCITT spec as whole integers.
  They are represented in a 16-bit fixed format as follows:

         integer        hexadecimal
         0        0000
         1        2000
         2        4000
         3        6000
         4        7FFF
         -1       E000
         -2       C000
         -3       A000
         -4       8000

  Registers used:
}

.VAR/DM     x_table[32];
.VAR/DM     y_table[32];

.INIT       x_table: H#8000, H#0000, H#0000, H#7FFF, H#7FFF,
                  H#0000, H#0000, H#8000, H#C000, H#C000, H#4000,
                  H#4000, H#4000, H#4000, H#C000, H#C000, H#A000,
                  H#2000, H#A000, H#2000, H#6000, H#E000, H#6000,
                  H#E000, H#2000, H#A000, H#2000, H#2000, H#E000,
                  H#6000, H#E000, H#E000;

.INIT       y_table: H#2000, H#A000, H#2000, H#2000, H#E000,
                  H#6000, H#E000, H#E000, H#6000, H#E000, H#6000,
                  H#E000, H#A000, H#2000, H#A000, H#2000, H#C000,
                  H#C000, H#4000, H#4000, H#4000, H#4000, H#C000,
                  H#C000, H#7FFF, H#0000, H#0000, H#8000, H#8000,
                  H#0000, H#0000, H#7FFF;
```

```
.EXTERNAL   conv_out, dac;
.ENTRY      sig_map;

sig_map:    I1=^x_table;
            I2=^y_table;

            M0=0;

            M1=DM(conv_out);
            MODIFY(I1,M1);
            MODIFY(I2,M1);

            AX0=DM(I1,M0);    {x value in ax0}
            AX1=DM(I2,M0);    {y value in ax1}

            DM(dac)=ax0;
            DM(dac)=ax1;

            RTS;
.ENDMOD;
```

Listing 2.7 Signal Mapping Routine

2.2.11 Viterbi Decoding

The V.32 recommendation specifies a trellis or convolutional encoding of data before transmission. The most common technique used for decoding received data is Viterbi decoding. The Viterbi algorithm is a general purpose technique for making an error-corrected decision. Viterbi decoding provides a certain degree of error correction by determining from the received bit pattern the value that was the most likely to have been transmitted. The Viterbi algorithm can be used for many applications where error correcting is required. Its application in the V.32 modem is similar to that used in other digital data communication schemes, such as digital telephones.

In order for the Viterbi algorithm to decode received data properly, the model for encoding the transmitted data must be known. In trellis encoding, it is assumed that the three delay elements of the encoder contain zeros initially. At each time period, a new 2-bit input is presented. The contents of the delay elements are changed accordingly and a 3-bit output is produced. If the three delay elements are treated as a 3-bit word, where delay element 1 is the most significant bit and delay element 3 is the least significant bit, then the state of the delay elements collectively can be represented by that 3-bit value.

It is possible to derive a state diagram or table from this specification. The three delay elements in the encoder are labelled from left to right as element 1, 2 and 3, respectively, in Figure 2.12. At any moment, each delay element has stored in it a 1 or a 0. The possible combinations of bits in the three delay elements or the possible states is eight. The state table shows the eight possible states of these three storage elements. It also shows that for any 2-bit input to the encoder, the three delay elements go to some new state and the encoder also produces an output. The state table showing the state transitions with the encoder inputs and outputs is shown in Table 2.2.

Beginning State	*Input*	*Output*	*End State*	*Beginning State*	*Input*	*Output*	*End State*
000	00	000	000	100	00	000	010
000	01	101	011	100	01	101	001
000	10	010	010	100	10	010	000
000	11	111	001	100	11	111	011
001	00	000	100	101	00	000	110
001	01	101	101	101	01	101	111
001	10	110	111	101	10	110	101
001	11	011	110	101	11	011	100
010	00	100	001	110	00	100	011
010	01	001	010	110	01	001	000
010	10	110	011	110	10	110	001
010	11	011	000	110	11	011	010
011	00	100	111	111	00	100	101
011	01	001	110	111	01	001	100
011	10	010	100	111	10	010	110
011	11	111	101	111	11	111	111

Table 2.2 State Table For Convolutional Encoder

Table 2.2 can also be used to derive a trellis diagram. The trellis diagram and the state diagram convey equivalent information. The trellis diagram for the convolutional encoder of the V.32 modem is shown in Figure 2.13.

Each node of the trellis represents a state and each node is labelled with the three-bit value of that particular state out of the eight possible states. A line is drawn from a state in one time window to a state of the next time window and represents the transition from one state to another for any given 2-bit input. Figure 2.13 shows some of the trellis paths labelled with the 3-bit output that was produced as the delay elements went from one state to another.

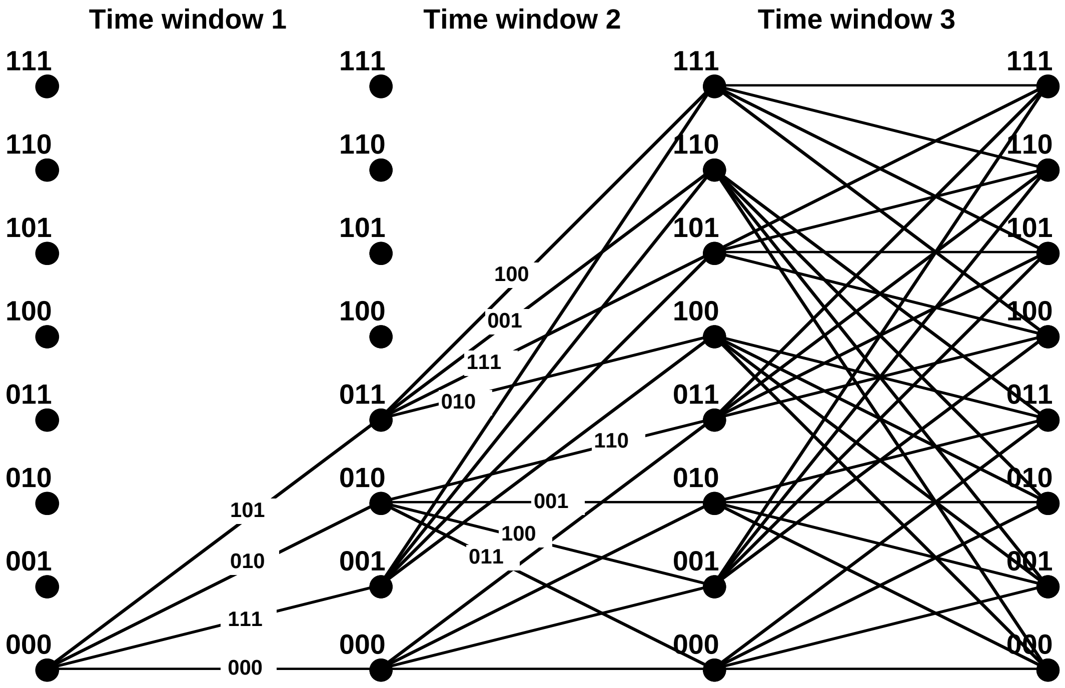

Figure 2.13 Trellis Diagram For Convolutional Encoding

It is assumed that at time t=0, the contents of each delay element is 0. Therefore the starting point for the trellis is at state 000. There are four possible combinations of 2-bit inputs and therefore, four lines that come out of state 000 and connect to the corresponding states at time window 2 as specified by the state table. For example, an input of 01 results in a change in the state of the delay elements from 000 to 011 with an output of 101. This information is conveyed in the trellis diagram by a line from state 000 to 011 labelled 101. The trellis diagram in Figure 2.13 has some of the branches labelled with the output value that is produced for a specific state transition; the rest can be determined from the state table.

2.2.12 Data Constellation

A 2-bit input to the convolutional encoder produces a 3-bit output containing a redundant bit. Because of redundancy, this 3-bit data value can be corrected for errors that occur during transmission.

In the transmission of information in a V.32 modem, the three bits from the output of the convolutional encoder are combined with two bits coming directly from the data bit stream. In essence, four bits from the data stream are being encoded to five bits (one redundant bit is added to the four original bits).

To modulate a carrier with this information, a constellation is created that maps any 5-bit data value to an X and Y coordinate or a real and imaginary term associated. The real and imaginary terms are used to modulate sine and cosine carriers for quadrature amplitude modulation. Figure 2.14 shows the V.32 constellation with the 3-bit output of the convolutional encoder underlined.

The demodulated carrier yields the original X and Y coordinates which determine the original 5-bit data value. Since the transmission medium for the carrier is noisy, the demodulated data may not be correct. The Viterbi algorithm corrects errors introduced in transmission.

2.2.13 Viterbi Algorithm

The Viterbi algorithm decides whether demodulated data is the data that was sent and if not, corrects it. It works by analyzing the pattern of data values received over a period of time to deduce the data value that is most likely to have occurred at the beginning of the period.

The received carrier is demodulated to produce X and Y coordinates of a point on the signal constellation. The distances from that point on the constellation to the nearest eight points that all have different leading three bits are calculated. These Euclidean distances are then used to label the branches of the trellis diagram. After a number of samples have been received and mapped to the trellis diagram in this fashion, the diagram can be read to determine the shortest path back to the original state, which determines the data value that has the highest probability of having been transmitted at that time.

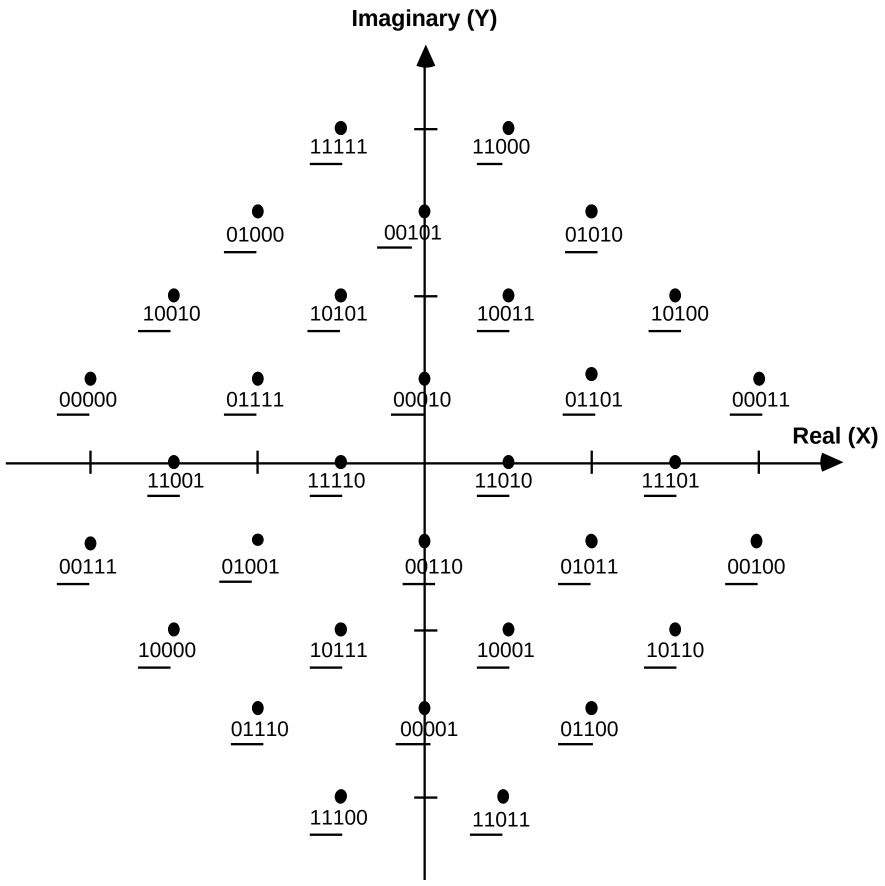

Figure 2.14 Signal Constellation Showing Convolutional Encoder Output

For example, assume that the received signal at time window 1 is mapped into the constellation at coordinate 2, 2 (x, y). This does not correspond to a five-bit code on the constellation. The Euclidean distances from this point to the nearest eight points are calculated. Because of the way the signal map is configured, each of these points has a different value for its first three bits (underlined in Figure 2.14).

In the trellis diagram, the line connecting state 000 to state 011 in time window 1 is labelled 101. The point in the signal constellation that is nearest to 2, 2 and has the value 101 as its first three bits is 10100, at coordinate 3, 2. The Euclidean distance between coordinate 2, 2 and 3, 2 is:

$$[(2–3)^2 + (2–2)^2]^{1/2} = 1$$

Therefore, the branch of the trellis diagram going from state 000 to state 011 is labelled 1. This process is repeated to label the other branches on the trellis diagram. As a new sample is received in each time window, the trellis branches are labelled with the corresponding Euclidean distances.

After a given number of time windows have elapsed, the shortest path back to the start of the first time window is calculated. The branch of the shortest path in the first time window represents the original data value that was transmitted.

Since the data point is determined only after a given number of time windows has elapsed, a delay of (number of time window multiplied by the symbol rate) is incurred. The more time windows that elapse before a decision is made, the more accurate the decision. Thus there is a tradeoff between accuracy and execution time.

2.2.14 ADSP-2100 Family Implementation

The first task of the program is to determine which eight points in the data constellation are the nearest to the X and Y coordinates produced by the demodulator. This is done using a lookup table. Each group in the lookup table contains the X and Y coordinates of the four points in the constellation that have the same 3-bit leading sequences. There are 32 points in the constellation, and therefore eight groups. Because the ADSP-2100 is a 16-bit machine, the X and Y values are normalized for 16-bit data. A negative full scale value of H#8000 and a positive full scale value of H#7FFF are used for both the X and Y values.

For example, 00000, 00001, 00010 and 00011 are in group 0. The Euclidean distance between the received point and the points in the group 0 are calculated. The shortest distance is then written into another table called *min_dist* in which the first location holds the shortest distance of the first group, the second location holds the shortest distance of the second group, etc. Table 2.3 shows the X and Y coordinates in each of the eight groups.

Group	*X*	*Y*	*Group*	*X*	*Y*
000	4	1	100	1	2
	0	1		–3	2
	–4	1		1	–2
	0	–3		–3	–2
001	4	–1	101	3	2
	0	–1		–1	2
	–4	–1		3	–2
	0	3		1	0
010	2	3	110	1	0
	–2	3		1	4
	2	–1		–3	0
	–2	–1		1	–4
011	2	1	111	3	0
	–2	1		–1	0
	2	–3		–1	4
	–2	–3		–1	–4

Table 2.3 Lookup Table Of X & Y Coordinates

2.2.15 Shortest Path Through Trellis Diagram

After the distance from the received point for the current time window to the closest point in each group is known, the total distance back to the beginning of the trellis diagram can be calculated. Each time, only the incremental distance for the time window, not the total distance, is calculated.

An 8-location table *acc_dist* stores the accumulated distance through the trellis diagram. Because the trellis diagram starts at state 000, the first location of the table is initialized with a 0 and all other locations with the positive full scale value. This ensures that, for the first time window, all paths converge back to state 000, since this state starts with the shortest accumulated distance.

At each time window, the surviving path to each state is determined and the accumulated distance table is updated with the accumulated distance of each of the eight surviving paths. The surviving path is determined by taking the length of all of the possible paths going into a state and adding that distance to the accumulated distance of the state at the other end of the path.

For example, Figure 2.15 shows the four paths that lead into state 001. The length of each path is added to the accumulated distance of the state from where the path emanates. The length of path 111 is added to the accumulated distance of state 000, the length of path 100 to is added the accumulated distance of state 010, the length of path 101 to is added the accumulated distance of state 100, and the length of path 110 is added to the accumulated distance of state 110. The lengths of these paths are read from the *min_dist* table.

The minimum of these four distances becomes the new accumulated distance to state 001 and is written into the appropriate location of the accumulated distance table (*acc_dist*). As each surviving path leg is determined, a table is filled with the distance of the path and the state from which it came, to allow the program to trace back along the surviving path to the beginning of the trellis diagram.

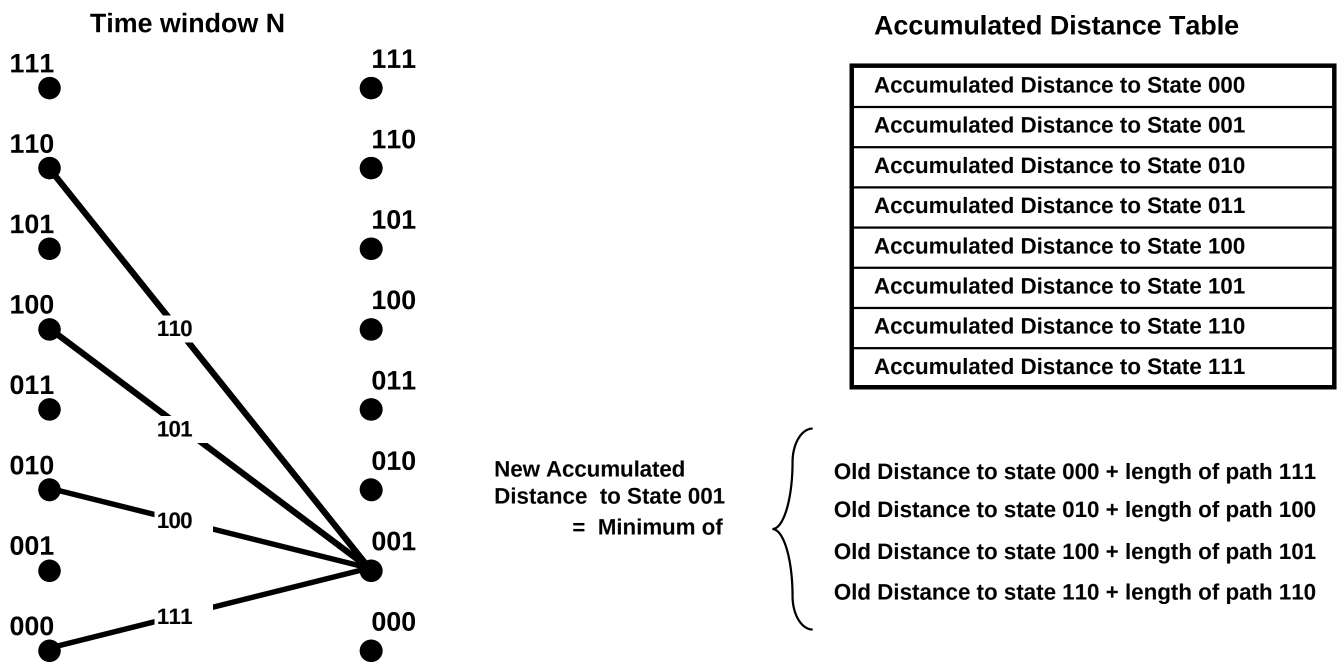
Time window N
111
110
101
100
011
010
001
000
110
101
100
111
Accumulated Distance Table
Accumulated Distance to State 000
Accumulated Distance to State 001
Accumulated Distance to State 010
Accumulated Distance to State 011
Accumulated Distance to State 100
Accumulated Distance to State 101
Accumulated Distance to State 110
Accumulated Distance to State 111
New Accumulated Distance to State 001 = Minimum of
Old Distance to state 000 + length of path 111
Old Distance to state 010 + length of path 100
Old Distance to state 100 + length of path 101
Old Distance to state 110 + length of path 110

Figure 2.15 Accumulated Distance Table Update Example

After all eight accumulated distances are updated, the shortest of the eight accumulated distances is determined. This path is traced back the given number of time windows. The distance of the branch in the first time window determines the data value most likely to have been transmitted. The point in the data constellation that is this distance from the received point represents the error-corrected symbol.

2.2.16 Viterbi Program

The example program uses N=20 time windows. In general, a value of N which is greater than or equal to three times the constraint length gives good results. In this case, the constraint length is 3, the number of bits needed to describe the possible states at each time window. The larger the value of N, the better the performance of the Viterbi algorithm, but the longer the execution time and the larger the table sizes.

2.2.16.1 Initialization

The first part of the program declares buffers and initializes variables. A buffer to store input data, eight tables holding the coordinates of the eight data groups, eight tables holding the 5-bit codes for the eight data groups, the accumulated distance buffer, eight state-tracing tables, eight buffers to hold surviving path distances and some pointer tables are all declared in the initialization section.

2.2.16.2 Data Input & Euclidean Distance

Data values are placed in registers AX0 and AX1 as X and Y coordinates, respectively, for input to the Viterbi program. The code starting at *find_dist* calculates the distances by calling the subroutine *dist* (which calculates the Euclidean distance squared) followed by the subroutine *sqrt*. This subroutine is repeated for each data group. The table *min_dist* is filled with the shortest distance for each group.

2.2.16.3 Shortest Path

The code starting at *short_path* determines the shortest surviving path to each state for the current time window. It also fills the eight state tables with the distance of the surviving branch and the state from which the branch came. The subroutine *min_calc* compares the four possible surviving paths and determine the shortest.

2.2.16.4 Last Surviving Path

After the accumulated distances to all eight states are calculated, the shortest is determined. The code starting at *search* determines the shortest path and traces this path back to the start of the trellis diagram.

2.2.16.5 Determination Of Error Corrected Data

When the surviving branch of the first time window is determined, the closest point of the data constellation in that data group is found. This 5-bit code is put into the SR1 register.

```
.MODULE/RAM       viterbi;

{Viterbi decoder program for convolutional encoded data for a V.32 modem. This
program decodes information using N=20 levels or time windows of Viterbi decoding.

Demodulated data is stored as input to this routine in registers AX0 and AX1 as
follows;

         AX0=X coordinate
         AX1=Y coordinate

This data is used as input.

The 5-bit data word output by this routine is placed in register SR1.}

.CONST            N=20;
.CONST            base=h#0D49, sqrt2=h#5A82;    {required for square root}
.VAR/PM/RAM       sqrt_coeff[5];
.INIT             sqrt_coeff: h#5D1D00, h#A9ED00, h#46D600,
                              h#DDAA00, h#072D00;


{table for storing last N inputs, as X and Y coordinate
table will contain alternating X, Y for each time window}

.VAR/DM/RAM/CIRC  inputs[N+N];

{variables to hold new X and Y inputs}
.VAR/DM/RAM       x_input;
.VAR/DM/RAM       y_input;
```

```
{tables for X and Y coordinates of data constellation points. Coordinates of both
axes are -4, -3, -2 ,-1, 0, 1, 2, 3, 4. They are represented in binary as:

         -4    H#8000
         -3    H#A000
         -2    H#C000
         -1    H#E000
         0     H#0000
         1     H#2000
         2     H#4000
         3     H#6000
         4     H#7FFF
}

.VAR/PM/RAM       group0[8];
.VAR/PM/RAM       group1[8];
.VAR/PM/RAM       group2[8];
.VAR/PM/RAM       group3[8];
.VAR/PM/RAM       group4[8];
.VAR/PM/RAM       group5[8];
.VAR/PM/RAM       group6[8];
.VAR/PM/RAM       group7[8];

.INIT group0:  H#7FFF00, H#200000, H#000000, H#200000,
                  H#800000, H#200000, H#000000, H#A00000;
.INIT group1:  H#7FFF00, H#E00000, H#000000, H#E00000,
                  H#800000, H#E00000, H#000000, H#600000;
.INIT group2:  H#400000, H#600000, H#C00000, H#600000,
                  H#400000, H#E00000, H#C00000, H#E00000;
.INIT group3:  H#400000, H#200000, H#C00000, H#200000,
                  H#400000, H#A00000, H#C00000, H#A00000;
.INIT group4:  H#200000, H#400000, H#A00000, H#400000,
                  H#200000, H#C00000, H#A00000, H#C00000;
.INIT group5:  H#600000, H#400000, H#E00000, H#400000,
                  H#600000, H#C00000, H#E00000, H#C00000;
.INIT group6:  H#200000, H#000000, H#200000, H#7FFF00,
                  H#A00000, H#000000, H#200000, H#800000;
.INIT group7:  H#600000, H#000000, H#E00000, H#000000,
                  H#E00000, H#7FFF00, H#E00000, H#800000;

{lookup table to get proper group}
.VAR/DM/RAM    group_table[8];

.INIT group_table:      ^group0, ^group1, ^group2, ^group3,
                        ^group4, ^group5, ^group6, ^group7;
```

(listing continues on next page)

```
{eight tables which show the 5-bit codes that correspond to the X and Y
coordinates in the 8 group tables}
.VAR/DM/RAM       codes0[4];
.VAR/DM/RAM       codes1[4];
.VAR/DM/RAM       codes2[4];
.VAR/DM/RAM       codes3[4];
.VAR/DM/RAM       codes4[4];
.VAR/DM/RAM       codes5[4];
.VAR/DM/RAM       codes6[4];
.VAR/DM/RAM       codes7[4];

.INIT codes0:  h#0003, h#0002, h#0000, h#0001;
.INIT codes1:  h#0004, h#0006, h#0007, h#0005;
.INIT codes2:  h#000A, h#0008, h#000B, h#0009;
.INIT codes3:  h#000D, h#000F, h#000C, h#000E;
.INIT codes4:  h#0013, h#0012, h#0011, h#0010;
.INIT codes5:  h#0014, h#0015, h#0016, h#0017;
.INIT codes6:  h#001A, h#0018, h#0019, h#001B;
.INIT codes7:  h#001D, h#001E, h#001F, h#001C;

.VAR/DM/RAM    codes_table[8];

.INIT codes_table:      ^codes0, ^codes1, ^codes2, ^codes3,
                        ^codes4, ^codes5, ^codes6, ^codes7;


{table for accumulated distances at each state}
.VAR/DM/RAM/CIRC  acc_dist[8];
.VAR/DM/RAM       temp_dist[8];

{eight tables where each table contains the possible states from where a
path could come for each of the eight states}

.VAR/DM/RAM       to_state0[4];
.VAR/DM/RAM       to_state1[4];
.VAR/DM/RAM       to_state2[4];
.VAR/DM/RAM       to_state3[4];
.VAR/DM/RAM       to_state4[4];
.VAR/DM/RAM       to_state5[4];
.VAR/DM/RAM       to_state6[4];
.VAR/DM/RAM       to_state7[4];
```

```
{table is stored with state numbers in backwards order}
.INIT to_state0:  2,4,6,0;
.INIT to_state1:  0,6,4,2;
.INIT to_state2:  6,0,2,4;
.INIT to_state3:  4,2,0,6;
.INIT to_state4:  5,3,7,1;
.INIT to_state5:  3,5,1,7;
.INIT to_state6:  1,7,3,5;
.INIT to_state7:  7,1,5,3;

{eight tables, each with N entries, where each entry contains the label of
the leg of the surviving path for a given time window}

.VAR/DM/RAM/CIRC  state0[N];
.VAR/DM/RAM/CIRC  state1[N];
.VAR/DM/RAM/CIRC  state2[N];
.VAR/DM/RAM/CIRC  state3[N];
.VAR/DM/RAM/CIRC  state4[N];
.VAR/DM/RAM/CIRC  state5[N];
.VAR/DM/RAM/CIRC  state6[N];
.VAR/DM/RAM/CIRC  state7[N];

{eight variables to hold the most recent pointer into the eight state
tables above}

.VAR/DM/RAM       pointer0;
.VAR/DM/RAM       pointer1;
.VAR/DM/RAM       pointer2;
.VAR/DM/RAM       pointer3;
.VAR/DM/RAM       pointer4;
.VAR/DM/RAM       pointer5;
.VAR/DM/RAM       pointer6;
.VAR/DM/RAM       pointer7;

.INIT pointer0:^state0;
.INIT pointer1:^state1;
.INIT pointer2:^state2;
.INIT pointer3:^state3;
.INIT pointer4:^state4;
.INIT pointer5:^state5;
.INIT pointer6:^state6;
.INIT pointer7:^state7;

{table used to look up pointers declared above}
.VAR/DM/RAM       point_table[8];
```

(listing continues on next page)

```
{initialize table with the addresses of the pointers}
.INIT point_table:      ^pointer0, ^pointer1, ^pointer2,
                        ^pointer3, ^pointer4, ^pointer5,
                        ^pointer6, ^pointer7;

{table to hold the eight possible distances, minimum of each group}
.VAR/DM/RAM    min_dist[8];


{interrupt vectors}
               RTI;
               RTI;
               RTI;
               JUMP decode;

               IMASK=0;       {disable all interrupts}
               ICNTL=8;       {interrupts edge sensitive, non-nested}
               ENA AR_SAT;

               I0=^inputs;    {init. I0 to start of input buffer}
               L0=%inputs;    {init. L0 to size of input buffer}
               M0=1;
               M1=0;
               M3=-1;

               L3=N;
               L5=0;

{initialize input buffer to all 0s}
               CNTR=%inputs;   {load counter with size of buffer}
               SI=0;           {put a 0 into register si}
               DO clear_buf UNTIL CE;
clear_buf:        DM(I0,M0)=SI; {transfer 0 into buffer location}

{initialize accumulated distance table}
               I1=^acc_dist;
               L1=%acc_dist;
               DM(I1,M0)=0;
               CNTR=%acc_dist-1;
               DO clear_acc UNTIL CE;
clear_acc:        DM(I1,M0)=h#7FFF;
```

```
{initialize eight tables with 0}
                I2=^state0;
                L2=%state0;
                CNTR=N;
                DO init_table0 UNTIL CE;
init_table0:       DM(I2,M0)=SI;

                I2=^state1;
                L2=%state1;
                CNTR=N;
                DO init_table1 UNTIL CE;
init_table1:       DM(I2,M0)=SI;

                I2=^state2;
                L2=%state2;
                CNTR=N;
                DO init_table2 UNTIL CE;
init_table2:       DM(I2,M0)=SI;

                I2=^state3;
                L2=%state3;
                CNTR=N;
                DO init_table3 UNTIL CE;
init_table3:       DM(I2,M0)=SI;

                I2=^state4;
                L2=%state4;
                CNTR=N;
                DO init_table4 UNTIL CE;
init_table4:       DM(I2,M0)=SI;

                I2=^state5;
                L2=%state5;
                CNTR=N;
                DO init_table5 UNTIL CE;
init_table5:       DM(I2,M0)=SI;

                I2=^state6;
                L2=%state6;
                CNTR=N;
                DO init_table6 UNTIL CE;
init_table6:       DM(I2,M0)=SI;

                I2=^state7;
                L2=%state7;
                CNTR=N;
                DO init_table7 UNTIL CE;
```

(listing continues on next page)

```
init_table7:      DM(I2,M0)=SI;

               L2=0;
               IMASK=8;            {enable interrupt 3}
waitlp:        JUMP waitlp;

{——————————————————————————————}

decode:        AX0=DM(codec);
               AX1=DM(codec);
               DM(I0,M0)=AX0;    {store X input in input buffer}
               DM(I0,M0)=AX1;    {store Y input in input buffer}
               DM(x_input)=AX0;
               DM(y_input)=AX1;


{Calculate Euclidean distances from received point to 32 points of data
constellation. The shortest distance in each data group is saved and will
represent the distance for the trellis branch for the current time window}

find_dist:     M4=1;
               L4=0;
               I4=^group0;
               CALL dist;
               AR=PASS AF;         {put distance squared into AR}
               MR0=0;
               MR1=AR;
               CALL sqrt;
               DM(min_dist)=SR1;   {store shortest dist in table}


               I4=^group1;
               CALL dist;
               AR=PASS AF;         {put distance squared into AR}
               MR0=0;
               MR1=AR;
               CALL sqrt;
               DM(min_dist+1)=SR1;  {store shortest dist in table}

               I4=^group2;
               CALL dist;
               AR=PASS AF;         {put distance squared into AR}
               MR0=0;
               MR1=AR;
               CALL sqrt;
               DM(min_dist+2)=SR1;  {store shortest dist in table}
```

```
        I4=^group3;
        CALL dist;
        AR=PASS AF;          {put distance squared into AR}
        MR0=0;
        MR1=AR;
        CALL sqrt;
        DM(min_dist+3)=SR1;  {store shortest dist in table}

        I4=^group4;
        CALL dist;
        AR=PASS AF;          {put distance squared into AR}
        MR0=0;
        MR1=AR;
        CALL sqrt;
        DM(min_dist+4)=SR1;  {store shortest dist in table}

        I4=^group5;
        CALL dist;
        AR=PASS AF;          {put distance squared into AR}
        MR0=0;
        MR1=AR;
        CALL sqrt;
        DM(min_dist+5)=SR1;  {store shortest dist in table}

        I4=^group6;
        CALL dist;
        AR=PASS AF;          {put distance squared into AR}
        MR0=0;
        MR1=AR;
        CALL sqrt;
        DM(min_dist+6)=SR1;  {store shortest dist in table}

        I4=^group7;
        CALL dist;
        AR=PASS AF;          {put distance squared into AR}
        MR0=0;
        MR1=AR;
        CALL sqrt;
        DM(min_dist+7)=SR1;  {store shortest dist in table}

        SR1=H#7fff;
        DM(min_dist+8)=SR1;

{Add each path distance to accumulated distance to yield 4 accumulated
distances for each state. The shortest accumulated distance becomes the new
accumulated distance to that state.}
```

(listing continues on next page)

```
{Find shortest path into state 0. Choose from 0, 1, 2, 3 of min_dist table; these
correspond to paths back to states 0, 6, 4, 2 respectively. The accumulated
distances to these states are added with the paths of the current time window to
determine the shortest accumulated path to this point.}

short_path:    I2=^min_dist;
               I3=^to_state0+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist)=AR;    {store temporarily}

               AX0=4;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}

{find the state from which the shortest path came}
               I2=^to_state0-1;
                              {point to 1 before start of table}
               M2=SI;         {get index into table}
               MODIFY(I2,M2); {point into table}
               SI=DM(I2,M1);  {get state at end of surviving path}

{now that state at end of path is known, store for later along with the 3-bit output
label of the suriving path; pack both into 1 word; state in high byte, label low
byte}

               SR=SR OR LSHIFT SI BY 8 (HI);
               I3=DM(pointer0);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer0)=I3;     {store new pointer}

{find shortest path into state 1, choose from 4, 5, 6, 7 of min_dist table these
correspond to paths back to states 2, 4, 6, 0 respectively}
               I2=^min_dist+4;
               I3=^to_state1+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+1)=AR;  {store temporarily}

               AX0=8;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}

{find the state from which the shortest path came.}
```

```
               I2=^to_state1-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,M1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the 3-
bit output label of the suriving path pack both into 1 word state is in high
byte, label lo byte.}

               SR=SR or LSHIFT SI BY 8 (HI);
               I3=DM(pointer1);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer1)=I3;     {store new pointer}



{find shortest path into state 2, choose from 0, 1, 2, 3 of min_dist table
these correspond to paths back to states 4, 2, 0, 6 respectively}
               I2=^min_dist;
               I3=^to_state2+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+2)=AR;  {store temporarily}

               AX0=4;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}

{find the state from which the shortest path came.}
               I2=^to_state2-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,I1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the 3-
bit output label of the suriving path pack both into 1 word state is in high
byte, label lo byte.}

               SR=SR or LSHIFT SI BY 8 (HI);
               I3=DM(pointer2);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer2)=i3;     {store new pointer}
```

(listing continues on next page)

```
{find shortest path into state 3, choose from 4, 5, 6, 7 of min_dist table
these correspond to paths back to states 6, 0, 2, 4 respectively}

               I2=^min_dist+4;
               I3=^to_state3+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+3)=AR;  {store temporarily}

               AX0=8;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}

{find the state from which the shortest path came.}
               I2=^to_state3-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,M1);        {get state at end of surviving path}


{now that state at end of path is known, store for later use along with the
3-bit output label of the suriving path pack both into 1 word state is in
high byte, label lo byte.}

               SR=SR OR LSHIFT SI BY 8 (HI);
               I3=DM(pointer3);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer3)=I3;     {store new pointer}


{find shortest path into state 4, choose from 0, 1, 2, 3 of min_dist table
these correspond to paths back to states 1, 7, 3, 5 respectively}

               I2=^min_dist;
               I3=^to_state4+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+4)=AR;  {store temporarily}

               AX0=4;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}
```

```
{find the state from which the shortest path came.}
               I2=^to_state4-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,M1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the
3-bit output label of the suriving path pack both into 1 word state is in
high byte, label lo byte.}

               SR=SR OR LSHIFT SI BY 8 (HI);
               I3=DM(pointer4);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer4)=I3;     {store new pointer}


{find shortest path into state 5, choose from 4, 5, 6, 7 of min_dist table
these correspond to paths back to states 7, 1, 5, 3 respectively}
               I2=^min_dist+4;
               I3=^to_state5+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+5)=AR;  {store temporarily}

               AX0=8;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, will pack later}

{find the state from which the shortest path came.}
               I2=^to_state5-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,M1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the
3-bit output label of the suriving path pack both into 1 word state is in
high byte, label lo byte.}

               SR=SR OR LSHIFT SI BY 8 (HI);
               I3=DM(pointer5);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer5)=I3;     {store new pointer}
```

(listing continues on next page)

```
{find shortest path into state 6, choose from 0, 1, 2, 3 of min_dist table
these correspond to paths back to states 5, 3, 7, 1 respectively}
               I2=^min_dist;
               I3=^to_state6+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+6)=AR;  {store temporarily}

               AX0=4;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}

{find the state from which the shortest path came.}
               I2=^to_state6-1;     {point to start of table}
               I2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,I1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the
3-bit output label of the suriving path pack both into 1 word state is in
high byte, label lo byte}
               SR=SR or LSHIFT SI BY 8 (HI);
               I3=DM(pointer6);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer6)=I3;     {store new pointer}


{find shortest path into state 7, choose from 4, 5, 6, 7 of min_dist table
these correspond to paths back to states 3, 5, 1, 7 respectively}
               I2=^min_dist+4;
               I3=^to_state7+3;
               CNTR=4;
               CALL min_calc;
               DM(temp_dist+7)=AR;  {store temporarily}

               AX0=8;
               AY0=SI;
               AR=AX0-AY0;    {calc. label from index of survivor}
               SR1=AR;        {store label into SR1, pack later}
```

```
{find the state from which the shortest path came.}
               I2=^to_state7-1;     {point to start of table}
               M2=SI;               {get index into table}
               MODIFY(I2,M2);       {point into table}
               SI=DM(I2,M1);        {get state at end of surviving path}

{now that state at end of path is known, store for later use along with the 3-bit
output label of the suriving path pack both into 1 word state is in high byte, label
lo byte.}

               SR=SR OR LSHIFT SI BY 8 (HI);
               I3=DM(pointer7);     {get pointer for state path}
               DM(I3,M0)=SR1;       {store state for current time window}
               DM(pointer7)=I3;     {store new pointer}

{Put data from temp_dist back into acc_dist as new accumulated distance up to this
point.}

replace:       CNTR=8;
               I2=^acc_dist;
               I1=^temp_dist;
               I1=0;
               DO move_buf UNTIL CE;
                  SI=DM(I1,M0);     {read data from temp_dist}
move_buf:         DM(I2,M0)=SI;     {put back as new acc_dist}

{Search through the acc_dist table for the shortest distance.  This will indicate
the end point of the surviving path.}

search:        I2=^acc_dist;
               CNTR=8;

               SI=CNTR;
               AY0=h#7FFF;          {initialize with largest number}
               AF=PASS AY0;
               AX0=DM(I2,M0);
               DO short_dst UNTIL CE;
                  AR=AF-AX0;
                  IF LE JUMP short_dst;
                  SI=CNTR;                {save index of smallest}
                  IF GE AF=PASS AX0;      {if smaller, update}
short_dst:        AX0=DM(I2,M0);

               AX0=8;
               AY0=SI;
               AR=AX0-AY0;          {calc. which state is at end of surviving path}
```

(listing continues on next page)

```
{Now that the end of surviving path is known (in AR), trace back N time
windows to find starting path or path of survivor in first time window.}

trace:          CNTR=N;               {trace back N time windows}
                DO search_back UNTIL CE;

{read entry from proper state table to find from which state path came}
                I2=^point_table;      {point to start of table}
                M2=AR;                {get offset into table}
                MODIFY(I2,M2);        {modify pointer to point into table}
                AX0=DM(I2,M1);        {read pointer address from table}

                I2=AX0;               {put pointer address into I2}
                AY1=DM(I2,M2);        {get pntr value, add. into state table}
                I2=AY1;
                AY0=N+1;              {calculate index into state table}
                AX0=CNTR;
                AR=AX0-AY0;
                M2=AR;
                L2=N;
                MODIFY(I2,M2);        {point into state table using circ}
                L2=0;
                SI=DM(I2,M1);         {read contents of state table}
                AX0=SI;
                AY0=h#FF;             {set up mask to isolate path label}
                AF=AX0 AND AY0;             {extract path label}

                SR=LSHIFT SI BY -8 (HI);   {extract state info}
search_back:    AR=SR1;

{At this point the surviving leg label is in AF and the state number in AR
find the 5-bit code in the group specified by value in AF that is closest
to the data recieved N time windows ago.}

final_stage:    AR=PASS AF;           {put leg label into AR}
                MX1=AR;               {store leg label in MX1,for later}
                I2=^group_table;      {point to start of group table}
                M2=AR;                {get displacement into table}
                MODIFY(I2,M2);        {update pointer}
                AX0=DM(I2,M1);        {get address of proper table}
                I4=AX0;               {load i4 with start of group table}

                AX0=DM(I0,M0);        {get X coord. of input N windows ago}
                M2=-1;
                AX1=DM(I0,M2);        {get Y coord. of input N windows ago}
                AY0=32767;                         {init with max distance}
                AF=PASS AY0, AY0=PM(I4,M4);        {get X value from table}
                CNTR=4;                            {4 points in group}
                DO ptloop2 UNTIL CE;
                   AR=AX0-AY0, AY1=PM(I4,M4);      {do X-X’ and get Y}
                   IF AV JUMP ptloop2;             {if overflow, go on}
                   MY0=AR, AR=AX1-AY1;             {copy X-X’, do Y-Y’}
                   IF AV JUMP ptloop2;             {if overflow, go on}
                   MY1=AR;                         {copy Y-Y’}
                   MR=AR*MY1(SS), MX0=MY0;         {square Y-Y’, copy X-X’}
                   MR=MR+MX0*MY0(RND);             {add square of X-X’}
                   AR=MR1-AF;                      {compare with previous}
                   IF GE JUMP ptloop2;             {if larger, no update}
                   AF=PASS MR1;                    {if smaller, update}
                   SI=CNTR;                        {save index of closest point}
ptloop2:           AY0=PM(I4,M4);                  {get next X value}

                AX0=4;
                AY0=SI;
                AR=AX0-AY0;            {calculate index from min pointer}
                I2=^codes_table;       {point to start of codes_table}
                M2=MX1;                {leg label is offset into table}
                MODIFY(I2,M2);
                SI=DM(I2,M1);          {get address of which codes buf}
                I2=SI;
                M2=AR;                 {get index into codes table}
                MODIFY(I2,M2);
                SR1=DM(I2,M1);         {get 5-bit code from table}

{SR1 now contains the answer}
answer:         DM(dac)=SR1;

         RTI;

{——————————— SUBROUTINES ————————————}

{Calculate the Euclidean distance squared between the point specified by the x
and y coordinates found data memory locations x_input and y_input and the
points specified by the x and y coordinates found in the table pointed to by
index register i4.  The index denoting the table entry which is closest to the
input point is left in register SI and the shortest distance squared is left in
register AF.}

dist:           AY0=32767;                     {init min distance to max num}
                AX0=DM(x_input);
                AX1=DM(y_input);
                AF=PASS AY0, AY0=PM(I4,M4);    {get X value from table}
                CNTR=4;                        {4 points in group}
```

(listing continues on next page)

```
               DO ptloop UNTIL CE;
                  AR=AX0-AY0, AY1=PM(I4,M4); {do X-X’ and get Y}
                  IF AV JUMP ptloop;         {if overflow, go on}
                  MY0=AR, AR=AX1-AY1;        {copy X-X’, do Y-Y’}
                  IF AV JUMP ptloop;         {if overflow, go on}
                  MY1=AR;                    {copy Y-Y’}
                  MR=AR*MY1(SS), MX0=MY0;    {square Y-Y’, copy X-X’}
                  IF MV SAT MR;
                  MR=MR+MX0*MY0(RND);        {add square of X-X’}
                  IF VM SAT MR;
                  AR=MR1-AF;                 {compare with previous}
                  IF GE JUMP ptloop;         {if larger, no update}
                  AF=PASS MR1;               {if smaller, update}
                  SI=CNTR;                   {save index of closest point}
ptloop:           AY0=PM(I4,M4);             {get next X value}
               RTS;

{——————————————————————————————}

{Take a 32-bit number whose most significant portion is in register MR1 and
least significant portion in register MR0 and calculate the 16-bit square
root. If the input is interpreted as a 16.16 unsigned number, the output in
register SR1 is in 8.8 signed format.}

sqrt:          I7=^sqrt_coeff;            {pointer to coeff. buffer}
               M4=1;
               L7=0;
               SE=EXP MR1(HI);            {check for redundant bits}
               SE=EXP MR0(LO);
               AX0=SE, SR=NORM MR1(HI);   {remove redundant bits}
               SR=SR OR NORM MR0(LO);
               MY0=SR1, AR=PASS SR1;
               IF EQ RTS;
               MR=0;
               MR1=base;                         {load constant value}
               MF=AR*MY0(RND), MX0=PM(I7,M4);    {MF = x squared}
               MR=MR+MX0*MY0(SS), MX0=PM(I7,M4); {MR = base + CX}
               CNTR=4;
               DO approx UNTIL CE;
                  MR=MR+MX0*MF(SS), MX0=PM(I7,M4);
approx:           MF=AR*MF(RND);
               AY0=15;
               MY0=MR1, AR=AX0+AY0;              {SE + 15 = 0?}
               IF NE JUMP scale;                 {no, compute square-root}
               SR=ASHIFT MR1 BY -7 (HI);
               RTS;
```

```
scale:          MR=0;
                MR1=sqrt2;            {load 1 over square rt of 2}
                MY1=MR1, AR=ABS AR;
                AY0=AR;
                AR=AY0-1;
                IF EQ JUMP pwr_ok;
                CNTR=AR;              {compute (1/sqr-rt 2)^(SE+15)}
                DO compute UNTIL CE;
compute:           MR=MR1*MY1(RND);
pwr_ok:         IF NEG JUMP frac;
                AY1=h#0080;           {load a 1 in 9.23 format}
                AY0=0;
                DIVS AY1, MR1;        {compute reciprocal MR}
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                DIVQ MR1;
                MX0=AY0;
                MR=0;
                MR0=h#2000;
                MR=MR+MX0*MY0(US);
                SR=ASHIFT MR1 BY 1(HI);
                SR=SR OR LSHIFT MR0 BY 1(LO);
                RTS;
frac:           MR=MR1*MY0(RND);
                SR=ASHIFT MR1 BY -7(HI);
                RTS;
```

(listing continues on next page)

```
{——————————————————————————————}

{Take the distances found in the table pointed to by register I2, add them
to the accumulated distance to the state specified in the state table
pointed to by register I3, and determine the shortest of these total
distances. The shortest distance is placed in register AR and the index of
the shortest distance is placed in register SI.}

min_calc:      L3=0;
               SI=CNTR;
               AY0=h#7FFF;                {initialize with largest number}
               AF=PASS AY0;

               MR1=DM(I2,M0);
               SR=ASHIFT MR1 BY -1(HI);   {half scale}
               AX0=SR1;

               DO short_dist UNTIL CE;

                  AY1=DM(I3,M3);          {read state number}
                  I5=^acc_dist;
                  M5=AY1;
                  MODIFY(I5,M5);          {point to proper acc_dist val}
                  MR1=DM(I5,M4);          {get acc_dist value}
                  AR=MR1-AY0;             {check for max value of acc_dist}
                  IF EQ JUMP read_nxt;          {if max go to next}
                  SR=ASHIFT MR1 BY -1(HI);      {half scale}
                  AY1=SR1;

                  AR=AX0+AY1;             {add new path to acc_dist}

                  AX0=AR;

                  AR=AF-AX0;
                  IF LE JUMP read_nxt;
                  SI=CNTR;                {save index of smallest}
                  IF GE AF=PASS AX0;      {if smaller, update}

read_nxt:         MR1=DM(I2,M0);
                  SR=ASHIFT MR1 BY -1(HI);      {half scale}
short_dist:       AX0=SR1;
               AX0=DM(I2,M0);
               AR=PASS AF;
               L3=N;
               RTS;

.ENDMOD;
```

Listing 2.8 Viterbi Decoder

2.3 QUADRATURE AMPLITUDE MODULATION

The CCITT V.32 modem recommendation calls for the use of quadrature amplitude modulation (QAM) in the transmit section and quadrature amplitude demodulation in the receive section of the modem. The encoded digital sequence to be transmitted is amplitude modulated in the digital domain and then converted to analog form (via a D/A converter) for transmission over the telephone wires. At the receiving end of the V.32 system, the received analog signal is digitized (via an A/D converter) and demodulated in the digital domain in order to recover the information that was sent.

This section describes the implementation of quadrature amplitude modulation and demodulation on the ADSP-2100 family of processors.

2.3.1 QAM Methodology

Double-sideband quadrature amplitude modulation (QAM) is a very efficient modulation technique in terms of bandwidth usage. In QAM, two quadrature (90° phase-shifted) carriers, cos $\omega_c k$ and sin $\omega_c k$, are amplitude-modulated by two separate information-bearing signals, as shown in Figure 2.16.

The synthesized digital sequence can be expressed as:

$$x(k) = m_1(k) \cos \omega_c k + m_2(k) \sin \omega_c k$$

where $m_1(k)$ and $m_2(k)$ are the two separate information-bearing signals. The QAM signal sequence x(k) has the spectrum:

$$X(2\pi F) = 1/2\, [M_1(\omega - \omega_c) + M_1(\omega + \omega_c)] - j\, 1/2\, [M_2(\omega - \omega_c) - M_2(\omega + \omega_c)]$$

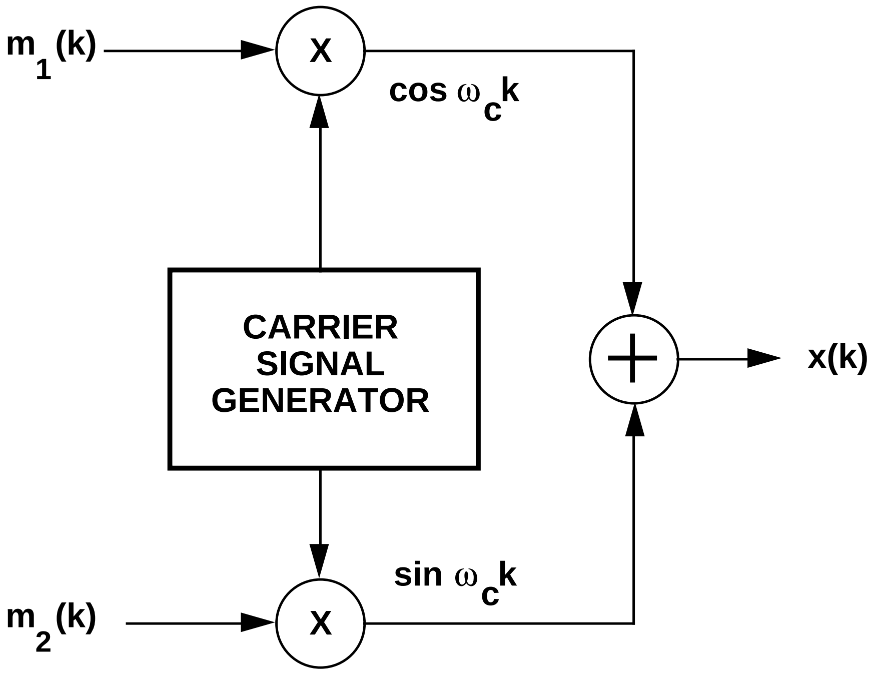

Figure 2.16 QAM Modulator Block Diagram

The spectrum components of the information-bearing signals overlap. However, the quadrature phase relationship in the carrier components cos $\omega_c k$ and sin $\omega_c k$ allows the receiving end of the V.32 system to separate the two signals.

The demodulation is performed as shown in Figure 2.17. A digital phase-locked loop is used to obtain the carrier component cos $\omega_c k$ and to generate sin $\omega_c k$.

Subsequently, the received sequence is multiplied by the two quadrature carriers. This multiplication results in two signal sequences:

$$x(k) \cos \omega_c k = 1/2\ m_1(k) + 1/2\ m_1(k) \cos 2\omega_c k + 1/2\ m_2(k) \sin 2\omega_c k$$

$$x(k) \sin \omega_c k = 1/2\ m_2(k) + 1/2\ m_2(k) \cos 2\omega_c k + 1/2\ m_1(k) \sin 2\omega_c k$$

The information-bearing signal components $m_1(k)$ and $m_2(k)$ can be recovered by passing each of the sequences through a filter that rejects the double-frequency terms centered at 2ω.

In this particular V.32 implementation, the carrier frequency (F_c) is 1800 Hz, the symbol rate is 2400 Hz and the sample rate of the modulator is 9600 Hz. Thus, the desired cosine carrier is:

$$\cos \omega_c k = \cos 2\pi F_c k T_s = \cos 2\pi(1800)(1/9600)\, k = \cos 3\pi/8\, k$$

and similarly the sine carrier is:

$$\sin \omega_c k = \sin 3\pi/8\, k$$

Again, in this particular V.32 implementation, the sequences $m_1(k)$ and $m_2(k)$ correspond to i(k) and q(k) respectively. These input streams are the filtered versions of quadrature and in-phase portions of the encoded symbols to be transmitted.

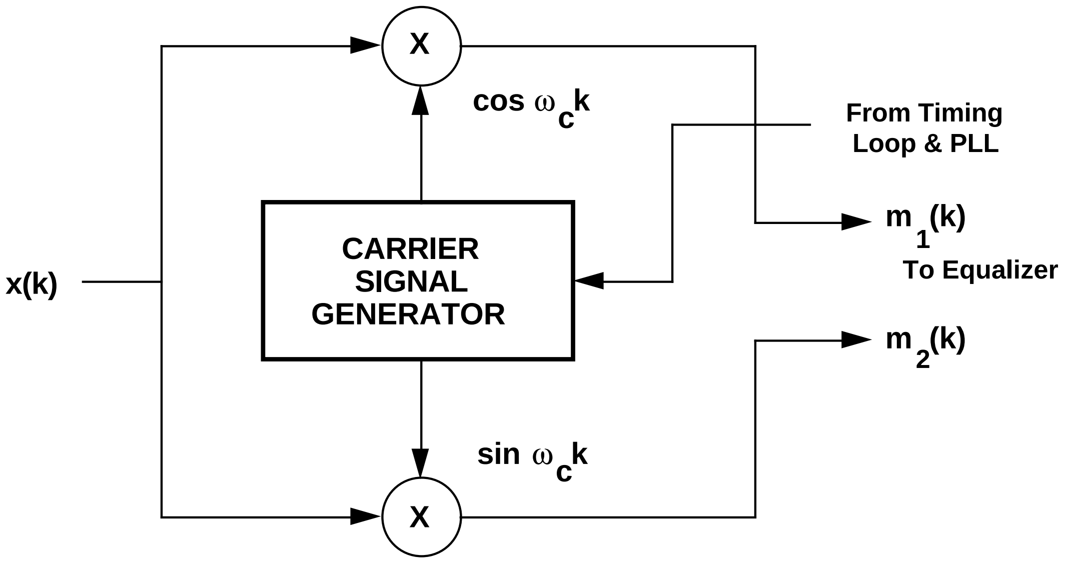

Figure 2.17 QAM Demodulator Block Diagram

2.3.2 ADSP-2100 Family Implementation

There are two ADSP-21XX assembly modules that handle the modulation and demodulation tasks separately. These modules are arranged as interrupt service routines that can be called from a main program which is presumably managing the V.32 modem.

Modulation is performed by the *modulator* routine shown on Listing 2.9. The first section of the code contains the necessary variable, constant and buffer declarations. The *cosine* table contains 16 discrete values of a cosine wave between 0 and 2π, in increments of π/8. This table is used to generate the cos3π/8k and sin3π/8k values for the modulation process. The variable *mod_ptr* stores a pointer into the *cosine* table between interrupts. The *mod_ptr* points to the cosine value to be modulated with the next arriving data sample.

```
.MODULE/RAM         modulator;
.VAR/PM/CIRC        cosine[16];             {Declare cosine table}
.VAR                cos_ptr;
.PORT               mod_out;

.INIT               cosine:<cosval.dat>;    {Initialize the cosine table}
.INIT               cos_ptr:^cosine;        {and the pointer}

.EXTERNAL           q_in, i_in;             {Input ports for i(k) and q(k)}
.GLOBAL             cosine, mod_out;
.ENTRY              modulate;

modulate:           I4=DM(cos_ptr);         {Read current pointer to cosine table}
                    M4=-4;
                    M5=7;
                    L4=16;
                    MX0=PM(I4,M4);          {Read current cos value}
                    MY0=DM(i_in);           {Read I(k)}
                    MR=MX0*MY0(SS),MX0=PM(I4,M5); {cos(k)*I(k) and get -sin value}
                    MY0=DM(q_in);                 {Read Q(k)}
                    MR=MR+MX0*MY0(RND);           {cos(k)*I(k)-sin(k)*Q(k)}
                    SR=ASHIFT MR2 BY -1(HI);      {Scale modulated output by 1/2}
                    SR=SR OR LSHIFT MR1 BY -1(LO);
                    DM(mod_out)=SR0;        {Send scaled output}
                    DM(cos_ptr)=I4;         {Save the cosine table pointer}
                    RTI;

.ENDMOD;
```

Listing 2.9 Modulator Code

The main body of the modulator code starts at the label *modulate*. The current cosine pointer is read and used to fetch the proper cosine value from the table. This fetch is done using M4=–4, which modifies the I4 register to point to the proper sine value on the following program memory (PM) fetch. Next, the i(k) input is read and multiplied with the cosine value. Subsequently, the proper sine value is fetched, multiplied with the q(k) input and added to the previous multiplication result. The sine value is fetched using M5=7 which modifies the I4 register to point to the proper cosine value on the following PM fetch. At this point, the MR register contains the output of the QAM modulator. Next, the contents of MR are scaled down by 1/2 using the shifter. This is necessary to keep the output of the modulator within a 16-bit field without causing overflows or underflows. Finally the current I4 value is saved as *mod_ptr* and the output is sent to the D/A converter.

The demodulation is handled by the *demodulator* routine shown in Listing 2.10. The first section of the code contains the necessary variable, constant and buffer declarations. This module also uses the *cosine* table that is declared and initialized in the modulator program. The variable *demod_ptr* points to the next cosine value for the demodulator, just as *mod_ptr* does for the modulator.

The main body of the demodulator code starts at the label *demodulate*. First, the current cosine pointer is read into I4. Next, the variable *phase_shift* is read in order to determine whether the phase-locked loop requires a phase shift in the cosine values to be used in demodulation. If a shift is required, the subroutine *cos_gen* is called to compute new values for the cosine table. Once this is completed, the appropriate cosine value is read from program memory using M4=–4. This value is multiplied with the input from the A/D converter and sent out to the memory location *xcos* which represents $x(k) \cos \omega_c k$. Subsequently, the proper sine value is fetched from program memory using M5=7 and multiplied with the A/D input. This result is sent to the memory location *xsin* which represents $x(k) \sin \omega_c k$. Finally, the current I4 value is saved as *demod_ptr*.

```
.MODULE/RAM       demodulator;
.VAR              cos_ptr;
.PORT             xsin;                   {Sine demodulated received signal}
.PORT             xcos;                   {Cosine demodulated received signal}
.PORT             ad_in;                  {Input port from the A/D}

.INIT             cos_ptr:^cosine;        {Initialize cosine table pointer}

.EXTERNAL         ph_shift_flag, cosine;
.GLOBAL           xsin, xcos;
.ENTRY            demodulate;

demodulate:       I4=DM(cos_ptr);         {Read current ptr to cosine table}
                  AY0=DM(ph_shift_flag);  {Read phase shift flag from the}
                                          {carrier recovery routine}
                  AR=PASS AY0;
                  IF NE CALL phase_shift; {Call if phase shift desired}
                  M4=-4;
                  M5=7;
                  L4=16;
                  MX0=PM(I4,M4);          {Read the current cosine value}
                  MY0=DM(ad_in);                      {Read the A/D input}
                  MR=MX0*MY0(RND),MX0=PM(I4,M5);      {cos(k)*x(k), get sine value}
                  DM(xcos)=MR1;           {Output cosine demodulated sample}
                  MR=MX0*MY0(RND);        {sin(k)*x(k)}
                  DM(xsin)=MR1;           {Output sine demodulated sample}
                  DM(cos_ptr)=I4;         {Save the cosine table pointer}
                  RTI;

phase_shift:      MODIFY(I4,M4);
                  MODIFY(I4,M5);
                  RTS;

.ENDMOD;
```

Listing 2.10 Demodulator Code

2.4 ECHO CANCELLATION

Most voiceband telephone connections involve several connections through the telephone network. The 2-wire subscriber line available at most sites is generally converted to a 4-wire signal at the telephone central office. The signal must be converted back to a 2-wire signal at the far-end subscriber line. The 2-to-4-wire interface is implemented with a circuit called a *hybrid*. The hybrid intentionally inserts impedance mismatches to prevent oscillations on the 4-wire trunk line. The mismatch forces a portion of the transmitted signal to be reflected or echoed back to the transmitter. This echo can corrupt data the transmitter receives from the far-end modem.

The telephone system and sources of echo are shown in Figure 2.18. There are two types of echo in a typical voiceband telephone connection. The first echo is the reflection from the near-end hybrid, and the second echo is from the far-end hybrid.

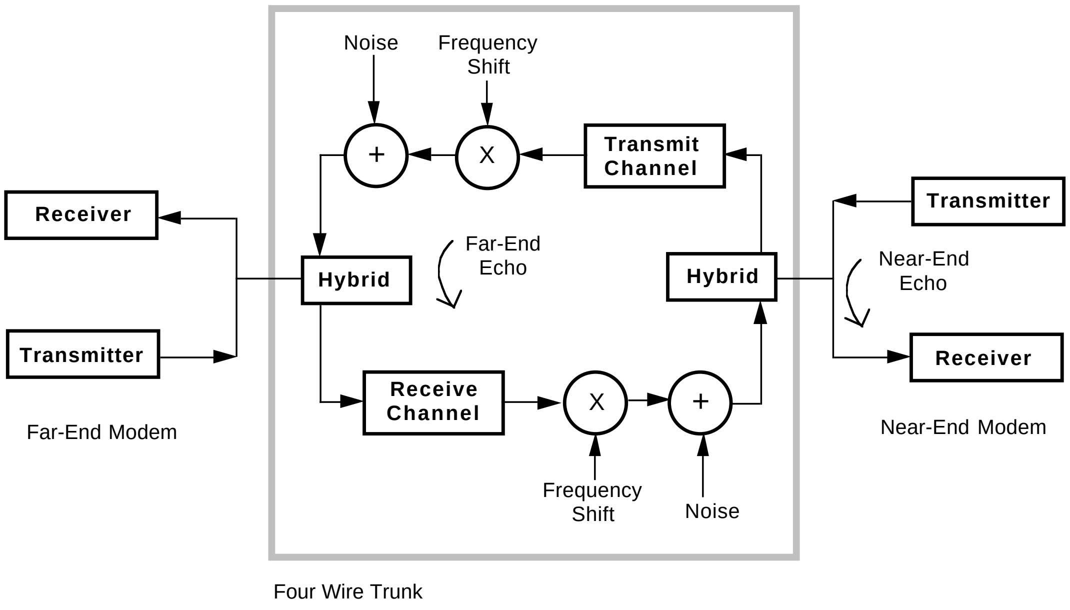

Figure 2.18 Telephone Channel Block Diagram

In long distance telephone transmissions, the transmitted signal is heterodyned to and from a carrier frequency. Since local oscillators in the network are not exactly matched, the carrier frequency of the far-end echo is offset from the frequency of the transmitted carrier signal. In modem applications this shift can affect the degree to which the echo signal can be cancelled. It is therefore desirable for the echo canceller to compensate for this frequency offset.

2.4.1 Echo Cancellation Algorithm

A data signal produced by a modem with a two-dimensional signal constellation has the form

$$s(t) = RE\ [\sum b_m g(t-mT)\ e^{j2\pi ft}]$$

where b_m is the complex data symbol and g(t) is the baseband pulse shape. The frequency *f* is the carrier frequency. The echo signal is the transmitted signal convolved with the channel transfer function, H(f). This transfer function usually involves a linear delay and some dispersive filtering. The echo signal has the form

$$s_e(t) = RE\ [\sum b_m h(t-mT)\ e^{j2\pi(f+f')t}]$$

where *f'* is the frequency offset (Weinstein, 1977).

If the near-end modem is transmitting a signal s(n) and the far-end modem is transmitting a signal y(n), the near-end received signal is:

$$r(n) = y(n) + s_{ne}(n) + s_{fe}(n) + w(n)$$

where s_{ne} and s_{fe} are the near-end and far-end echo respectively, and w(n) is random noise introduced by the system.

Echo cancellation is accomplished by subtracting an estimate of the echo return signal from the actual received signal. The received signal after echo cancellation is

$$r'(n) = y(n) + (s_{ne}(n) - \hat{s}_{ne}(n)) + (s_{fe}(n) - \hat{s}_{fe}(n)) + w(n)$$

where $\hat{s}_{fe}(n)$ is the estimate of the far-end echo and $\hat{s}_{ne}(n)$ is the estimate of the near-end echo. Ideally, the estimates are equal to the echo signals and the echo terms drop out (Quatieri and O'Leary, 1989).

The estimated echo is generated by feeding the transmitted signal into an adaptive filter whose transfer function tries to model the telephone channel's (see Figure 2.19). The filter coefficients are determined using the stochastic gradient (Least Mean Squared, or LMS) algorithm (Kamilo and Messerschmitt, 1987) during a training sequence prior to full duplex communications. The LMS algorithm attempts to minimize the mean squared error $|E(n)^2|$. A more detailed description of the LMS algorithm can be found later in this chapter.

In the training sequence, because the far-end modem is not transmitting, the received signal consists of echo:

$r(n) = s_{ne}(n) + s_{fe}(n)$

The output of the filter is an estimate of the received signal,

$r\wedge(n) = \wedge s_{ne}(n) + \wedge s_{fe}(n)$

and the difference is the error term that the LMS algorithm operates on.

$E(n) = r(n) – r\wedge(n)$

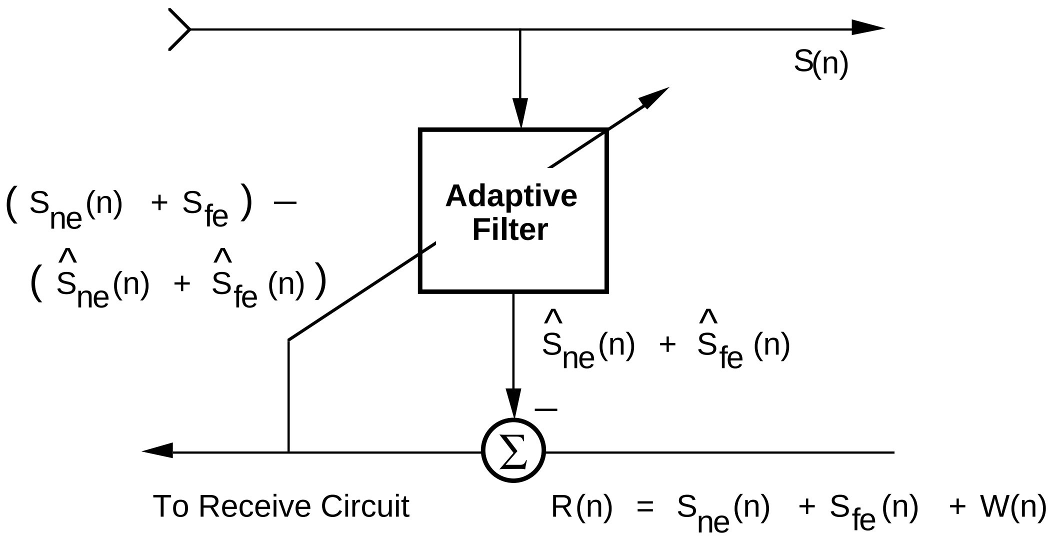
S(n)
Adaptive Filter
$(S_{ne}(n) + S_{fe}) -$
$(\hat{S}_{ne}(n) + \hat{S}_{fe}(n))$
$\hat{S}_{ne}(n) + \hat{S}_{fe}(n)$
$-$
Σ
To Receive Circuit
$R(n) = S_{ne}(n) + S_{fe}(n) + W(n)$

Figure 2.19 Echo Canceller

The adaptive filter is commonly implemented with a transverse FIR filter. The structure of this filter is shown in Figure 2.20. The LMS update equation for tap *C* at sample time *n* is

$$C(n)_{k+1} = C(n)_k + \beta A(n)E(n)$$

where A(n) is the sample transmitted at sample time *n*, E(n) is the residual error and β is an adaptation constant related to the rate of convergence.

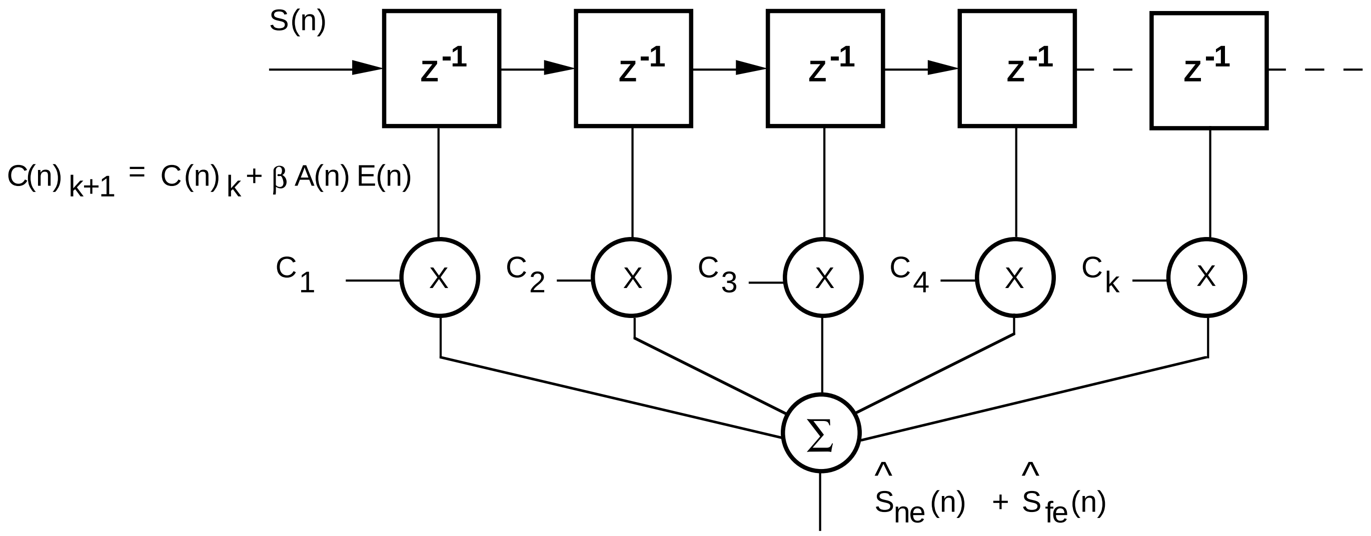

Figure 2.20 LMS Adaptive Filter

In a modem application, the filter taps are only updated during the training periods. The tap update algorithm is either disabled or the adaptation constant β is greatly reduced during full duplex operation. In the second case, reducing β allows the echo canceller to track a slowly changing telephone channel without retraining the modem.

2.4.2 ADSP-2100 Family Implementation Of LMS Algorithm

Figure 2.21 shows a flowchart for implementing the LMS stochastic gradient algorithm on the ADSP-2100 family of processors. The LMS algorithm is implemented in an interrupt service routine so that the arrival of a new sample forces one iteration of the algorithm. In this example, the FIR filter and the tap update are implemented as subroutine calls from the interrupt service routine.

In applications such as V.32 modems, the tap update algorithm gets disabled during full duplex operation.

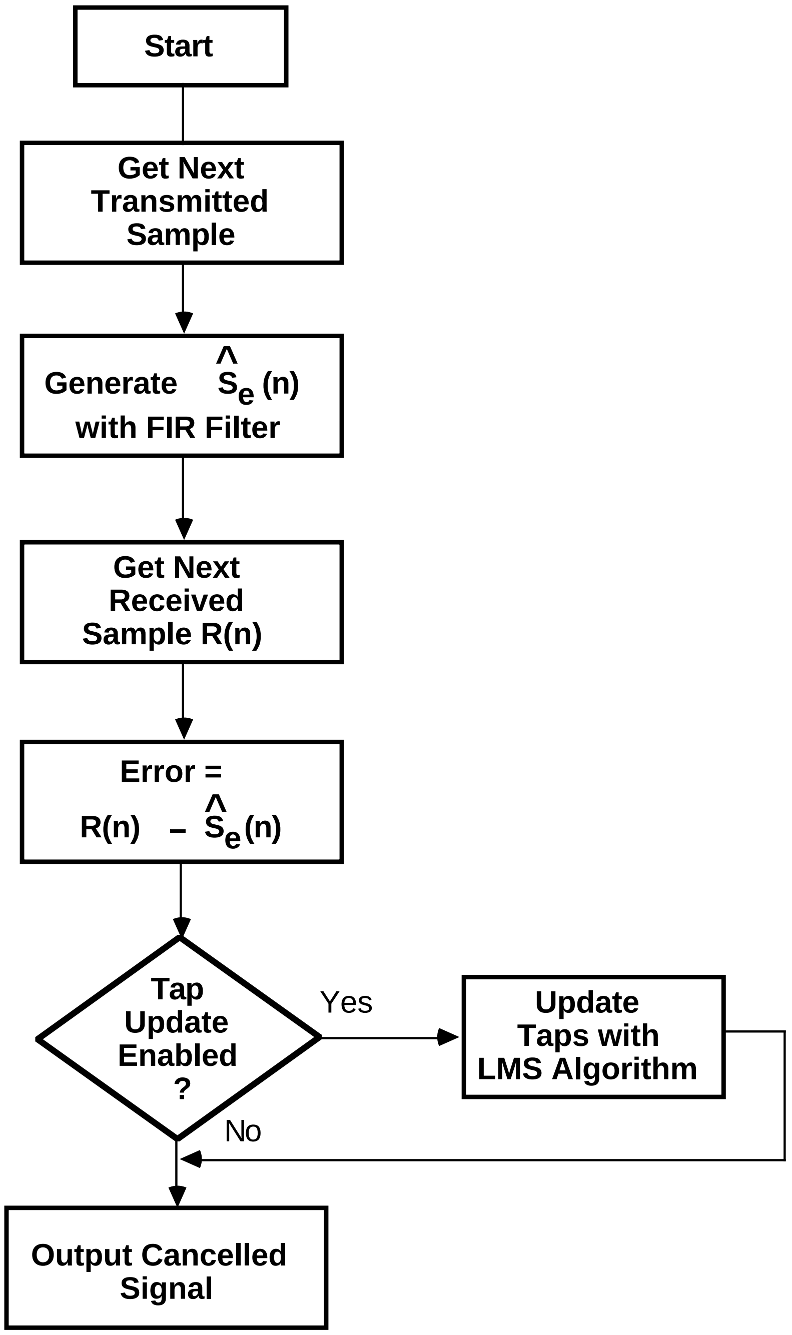
Start
Get Next Transmitted Sample
Generate $\hat{S}_e(n)$ with FIR Filter
Get Next Received Sample R(n)
Error = $R(n) - \hat{S}_e(n)$
Tap Update Enabled ?
Yes
Update Taps with LMS Algorithm
No
Output Cancelled Signal

Figure 2.21 Flowchart For LMS Stochastic Gradient Algorithm

Listing 2.11 contains the LMS filter code. The ADSP-2100 family can execute a multiply / accumulate operation and fetch two operands in a single cycle. The FIR filter loop and the tap update loop are executed without any additional cycles for loop overhead. These features allow the FIR filter to execute in one cycle per tap and the coefficient update to execute in two cycles per tap. Table 2.4 summarizes the execution speeds.

Some applications require the echo canceller to operate on complex data. A complex data implementation of the LMS algorithm is described later in this chapter.

```
.MODULE/RAM/ABS=0    adaptive;

{ Near and Far End Echo Canceller
                     INPUT: Received Data from Channel
                     Transmitted Data
                     OUTPUT: To Rest of Modem
}

.PORT     received_data;            {Received sample from channel}
.PORT     transmitted_data;         {Transmitted sample from modem}
.PORT     out;                      {Output to rest of modem}

.CONST               A=154;                      {Adaptive filter length}
.CONST               beta=H#CC;                  {Adaptation constant}
.VAR/DM/RAM/CIRC     enable;                     {Update enabled bit}
.VAR/DM/RAM/CIRC     afilt_data[A];              {Filter delay line}
.VAR/PM/RAM/CIRC     afilt_coeff[A];             {Filter coefficients}

{ Each new sample asserts interrupt 3}
start:    RTI;
          RTI;
          RTI;
          JUMP sample;
```

```
{ Initialize Routine: This is executed during system startup}
.ENTRY    setup;

setup:    ICNTL=B#01111;            {Initialize Interrupts}
          M0=0;                     {Initialize DAGS}
          M1=1;
          M3=-1;
          M4=1;
          M5=1;
          M6=-1;
          M7=2;
          I0=^afilt_data;
          I4=^afilt_coeff;
          L0=%afilt_data;
          L4=%afilt_coeff;
          AX0=H#0000;
          AY1=H#0000;               {Initialize filter to 0}
          CNTR=%afilt_data;
          DO foo3 UNTIL CE;
foo3:        PM(I4,M4)=AY1,DM(I0,M1)=AX0;
          IMASK=B#1000;             {Enable IRQ2}
fevr:     JUMP fevr;                {Wait for Interrupt}

{ Interrupt Routine: This code processes one data sample}
sample:   AY0=DM(received_data);    {Received data: r(n)}
          SR0=DM(transmitted_data); {Transmitted data: A(n)}
          CALL fir;                 {Calculate r^(n)}
          AR=AY0-MR1;               {AR=error=r-r^}
          DM(out)=AR;               {Output cancelled data}
          AX0=DM(enable);           {Update taps if enabled}
          AF=PASS AX0;
          IF EQ CALL update;
done:     RTI;

{ FIR Filter
          INPUTS:
             I0=Start of data buffer in DM
             I4=Start of coeff buffer in PM
             SR0=Newest input value
             M1,M4=1
          OUTPUTS:
             MR=Output value
          ALTERS:
             MR, MY0, MX0
}
```

(listing continues on next page)

```
.ENTRY    fir;

fir:      DM(I0,M1)=SR0;
          MR=0, MX0=DM(I0,M1), MY0=PM(I4,M4);
          CNTR=A-1;
          DO floop UNTIL CE;
floop:       MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
          MR=MR+MX0*MY0(RND);
          RTS;

{ Adaptive Filter Coefficient Update
          INPUTS:
             I0=Start of data buffer in DM
             I4=Start of coeff buffer in PM
             M1,M4=1
             M6=-1
             M7=+2
             AR=error of last iteration

  Executes the coeff update algorithm as follows:
          Ck+1=Ck+Beta*Error*A(n)
}
.ENTRY    update;

update:   MY1=beta;                        {Load Beta}
                                           {MF=Beta*Error, Load Ck, A(n)}
          MF=AR*MY1(RND), AY0=PM(I4,M4), MX0=DM(I0,M1);
          MR=MX0*MF(RND);
          CNTR=A;                          {Tap update loop}
          DO uloop UNTIL CE;
             AR=MR1+AY0, AY0=PM(I4,M6), MX0=DM(I0,M1);
uloop:       PM(I4,M7)=AR, MR=MX0*MF(RND);
          MODIFY(I0,M3);
          MODIFY(I4,M6);
          RTS;
.ENDMOD;
```

Listing 2.11 LMS Stochastic Gradient Implementation

2.4.3 Frequency Offset Compensation

Frequency offset in the far-end echo can limit convergence of the adaptive filter. In order to compensate for shifts in the carrier frequency, it is necessary to shift the received signal back to the original carrier frequency. Figure 2.22 shows a block diagram for performing this operation. The frequency shifter is a first-order digital phase locked loop (DPLL). The magnitude of the frequency shift is defined as

Ø^(n+1) = Ø^(n) + β A(n) (Ø(n) – Ø^(n)) r(n)

where β is the adaptation constant, Ø(n) is the frequency offset of sample *n*, Ø^(n) is the estimate of the frequency offset, A(n) is the transmitted sample, and r(n) is the received sample from the echo channel (Wang and Werner, 1988).

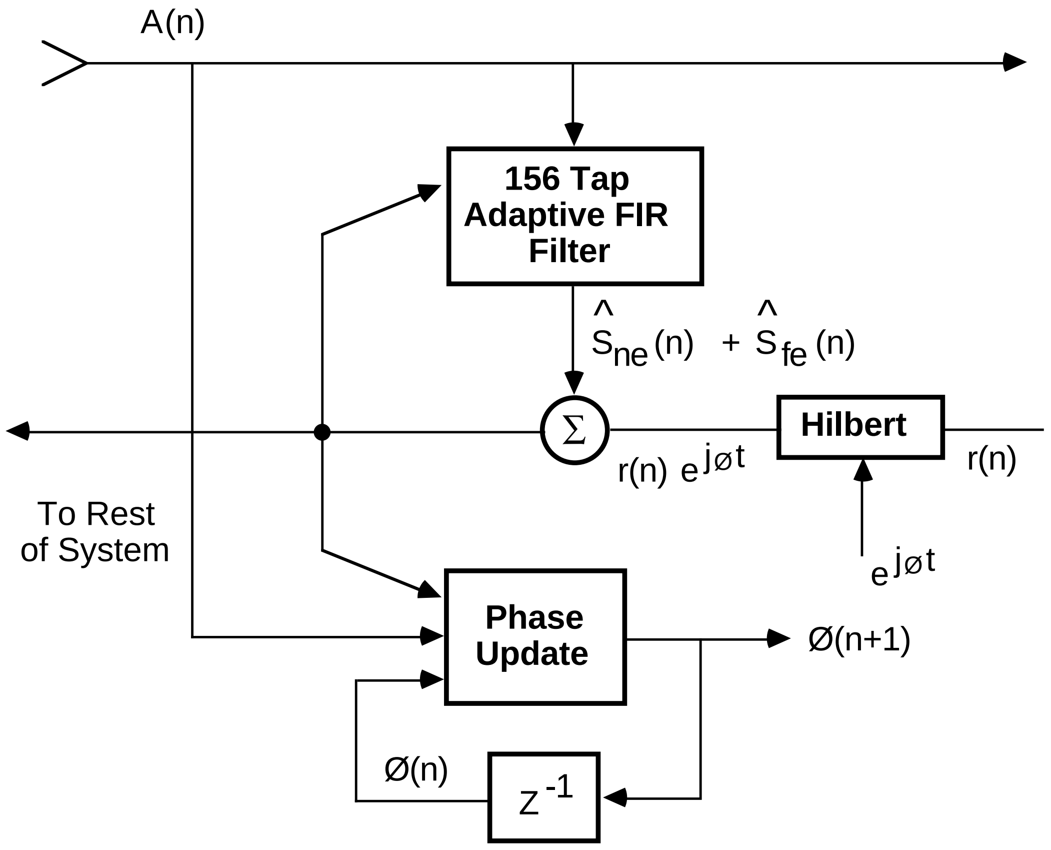

Figure 2.22 Block Diagram Of Echo Canceller With Frequency Shift

When compensating for frequency offset, the received sample must be rotated before the error term is calculated. The new error equation is

$E(n) = r(n)\,e^{j\varnothing t} - r(n)^\wedge$

In a real system, the frequency shift is implemented in the time domain with a Hilbert transform algorithm. Figure 2.23 shows the general structure of this algorithm.

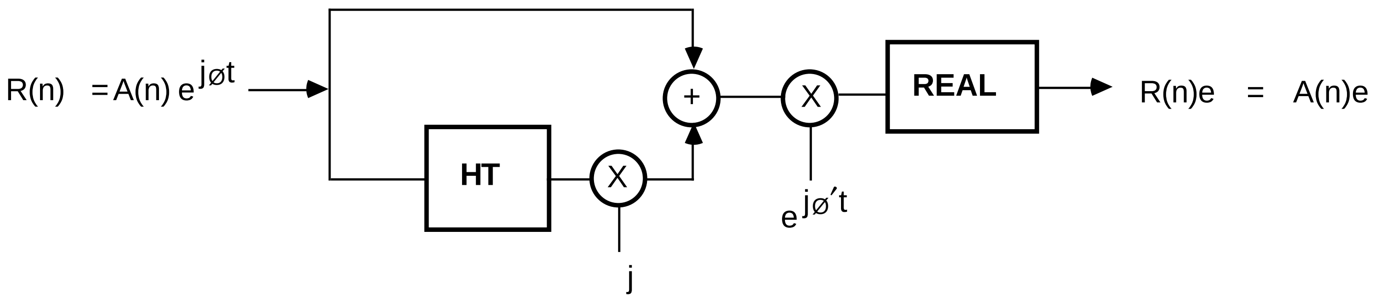

Figure 2.23 Block Diagram Of Hilbert Transform

The Hilbert algorithm is best understood in the frequency domain. Consider the real, bandlimited signal shown in Figure 2.24a. The Hilbert transfer function is

$$\begin{aligned} H(\omega) &= -j \quad \omega > 0 \\ &= +j \quad \omega < 0 \end{aligned}$$

The output of the Hilbert transform is multiplied by +j so that the frequency magnitude is real. The sum of the Hilbert transform and the original sample is complex in the time domain and contains only positive frequencies in the frequency domain. The magnitude in the frequency domain is equal to twice the magnitude of the original sample (Figure 2.24d).

The frequency shift is accomplished by convolving (in the frequency domain) the signal in Figure 2.24d with the desired frequency. This convolution is equivalent to multiplying the time domain signal by $e^{-j\omega_o t}$, where ω_o is the desired frequency shift. The sample is converted back to a real signal by taking the real part of the complex waveform.

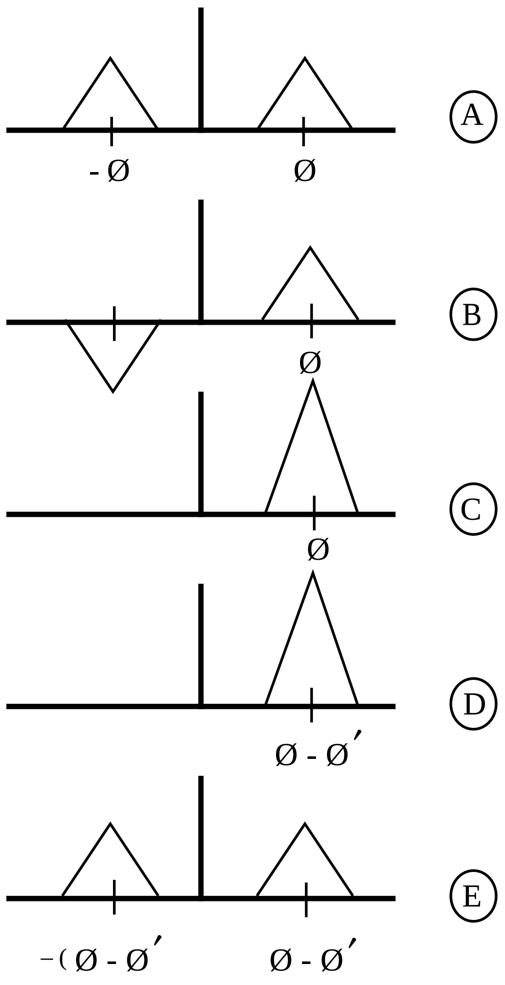

Figure 2.24 Spectrum Of Hilbert Frequency Shift

2.4.4 ADSP-2100 Family Implementation Of Hilbert Transform

Code implementing a Hilbert transform is shown in Listing 2.12. The received signal must be rotated before E_n, the error signal for the adaptive filter, can be calculated. The Hilbert transform is thus performed in a subroutine called from the LMS interrupt service routine.

The Hilbert transform is implemented with a 31-tap transverse FIR filter. Since every other coefficient is zero, the circular buffers in the ADSP-2100 are programmed to access every other data sample. This is possible using multiple modify registers with a single index register in the data address generators. The 31-tap Hilbert transform executes in 20 cycles.

To compensate for the group delay in the Hilbert transform, a 15-cycle linear delay is required for the real-valued input signal. Again, the circular buffering capabilities of the ADSP-2100 family allow for a simple implementation. Once the delay line is initialized, the index registers automatically increment to the next value, even when the end of the buffer is reached. The 15-tap delay line executes in just 3 cycles per sample.

The addition operation described shown in Figure 2.23 is actually summing of a real and a complex number. Since a real and imaginary number cannot be added, this operation is not implemented in the code. Instead, the real and imaginary parts are used in the complex multiplication.

The complex multiply by $e^{-j\omega_o t}$ would normally require four multiplications and two additions. In practice, the desired output is contained entirely in the real part of the product. Therefore, only two multiplications and one addition are required. The values for $\sin(\omega_o t)$ and $\cos(\omega_o t)$ must be calculated for each successive sample.

The single cycle multiply / accumulate operation on the ADSP-2100 family allows both multiplications and the addition to be executed in two cycles. Execution time is also reduced when operands are fetched from data memory in parallel with the multiplications. In transmit mode, the entire Hilbert frequency shift requires about 100 cycles to execute.

```
.MODULE/RAM/ABS=0       hilbert_rotator;

{ Hilbert Rotator
         INPUT: Received Sample
         OUTPUT: To Adaptive Filter
}

.CONST                  H=31;                 {Length of Hilbert xform filter}
.PORT                   received_data;        {Received sample from channel}
.PORT                   out;                  {Output to rest of modem}
.VAR/DM/RAM/CIRC        hdelay[H];            {Delay line for phase matching}
.VAR/DM/RAM/CIRC        hil_dat[H];           {filter data values}
.VAR/PM/RAM/CIRC        hilbert_coeff[16];    {Hilbert filter coefficients}
.VAR/DM/RAM             time;
.VAR/DM/RAM             delta_time;           {Delta for frequency shift}
.VAR/DM/RAM             high;
.VAR/DM/RAM             low;
.VAR/DM/RAM             ovr;

.INIT                   hilbert_coeff: <hilb.dat>;
                                              {Hilbert filter coefficients}

{ Initialize Routine: This is executed during system startup}
.ENTRY   setup;

setup:   AX0=H#00;
         DM(time)=AX0;
         AX0=H#02;
         DM(delta_time)=AX0;
         CNTR=^HIL_DAT;             {Init Delay line, Hilbert data}
         DO iloop UNTIL CE;
            DM(I0,M1)=H#0000;
iloop:      DM(I1,M1)=H#0000;
         IMASK=B#1000;              {Enable IRQ2}
fevr:    JUMP fevr;                 {Wait for Interrupt}

{ Interrupt Routine: This code processes one data sample}
sample:  AY0=DM(received_data);     {Received data: r(n)}
         CALL delay;                {Insert r(n) into delay line}
         CALL hilb;                 {Execute Hilbert transform}
         CALL rotate2;
         AR=MR1;
         DM(out)=AR;
         RTI;
```

(listing continues on next page)

```
{ 31 Tap Linear Delay Line
         INPUTS:        AY0=Newest Input Value
                        I0=Oldest value in delay
                        M0=0
                        M1=1
         OUTPUTS:       AX1=Delay line output
}
.ENTRY   delay;

delay:   AX1=DM(I0,M0);
         DM(I0,M1)=AY0;
         RTS;

{ 31 Tap Fir Hilbert Filter
         INPUTS:        AY0=Newest Input Data
                        I1=Oldest data value
                        I4=First Coeff value
                        M0=0
                        M1=1
                        M4=1
         OUTPUTS:       AY0=Hilbert output
}
.ENTRY   hilb;

hilb:    MR=0, MX0=DM(I1,M2), MY0=PM(I4,M4);
         CNTR=16;
         DO hil_loop UNTIL CE;
hil_loop:     MR=MR+MX0*MY0(SS), MX0=DM(I1,M2), MY0=PM(I4,M4);
         MR=MR+MX0*MY0(RND);
         DM(I1,M1)=AY0;
         AY0=MR1;
         RTS;

{ Hilbert Rotator
         Perform the calculation:
            Y(t)=RE[(Xr(t)+jXi(t)*(exp(-jWt))]

         INPUTS:        AY0=Xi(t)
                        AX1=Xr(t)
                        AY1=W in degrees-q15 format
                        W*t=DM(time)=time in q15
         OUTPUTS:       MR=Y(t)
}
```

```
.ENTRY   rotate2;

rotate2: AX0=DM(time);                 {Get and update rotate time}
         AY1=DM(delta_time);           {on unit circle}
         AR=AX0+AY1, MY0=AY0;          {MY0=im(x)}
         IF AC AR=PASS 0;
         DM(time)=AR;
         CALL sin;                     {Xi(t)*IM[exp(-jwt)]}
         MR=AR*MY0(SS), MY0=AX1;
         DM(ovr)=MR2;
         DM(high)=MR1;
         DM(low)=MR0;
         AY0=H#4000;                   {Xr(t)*sin(wt+90)}
         AR=AX0+AY0;
         AX0=AR;
         CALL sin;
         MR0=DM(low);
         MR1=DM(high);
         MR2=DM(ovr);
         MR=MR+AR*MY0(RND);
         RTS;

{ Sine Calculation
         Sine Approximation:  Y=Sin(x)

         INPUTS:        AX0=x in scaled 1.15 format
                        M3=1
                        L3=0
         OUTPUTS:       AR=y in 2.14 format

         Computation Time: 25 cycles
}
```

(listing continues on next page)

```
.VAR/DM  sin_coeff[5];
.INIT    sin_coeff: H#3240, H#0053, H#AACC, H#08B7, H#1CCE;
.ENTRY   sin;

sin:     I3=^sin_coeff;             {Pointer to coeff. buffer}
         AY0=H#4000;
         AR=AX0, AF=AX0 AND AY0;    {Check 2nd or 4th quad}
         IF NE AR=-AX0;             {If yes, negate input}
         AY0=H#7FFF;
         AR=AR AND AY0;             {Remove sign bit}
         MY1=AR;
         MF=AR*MY1(RND), MX1=DM(I3,M3);         {MF=x2}
         MR=MX1*MY1(SS), MX1=DM(I3,M3);         {MR=C1x}
         CNTR=3;
         DO approx UNTIL CE;
            MR=MR+MX1*MF(SS);
approx:     MF=AR*MF(RND), MX1=DM(I3,M3);
         MR=MR+MX1*MF(SS);
         SR=ASHIFT MR1 BY 2(HI);
         SR=SR OR LSHIFT MR0 BY 2(LO);  {Convert to 2.14 format}
         AR=PASS SR1;
         IF LT AR=PASS AY0;         {Saturate if needed}
         AF=PASS AX0;
         IF LT AR=-AR;              {Negate output if needed}
         RTS;
.ENDMOD;
```

Listing 2.12 Hilbert Transform Implementation

2.4.5 V.32 Modem Implementation

V.32 modems operate in full duplex mode; both the near-end and far-end modem are transmitting data at the same time. The echo canceller is responsible for channel separation as well as cancelling the near-end and far-end echos.

The echo canceller can be implemented in the passband or the baseband. The advantage of passband cancellation is reduced computation. A baseband echo canceller must execute all algorithms on complex data. In addition, compensating for frequency shift in the baseband is difficult. The disadvantage of passband echo canceller is a longer convergence time for the adaptive filter and the digital phase locked loop. Figure 2.25 shows a block diagram of a V.32 modem with a passband echo canceller.

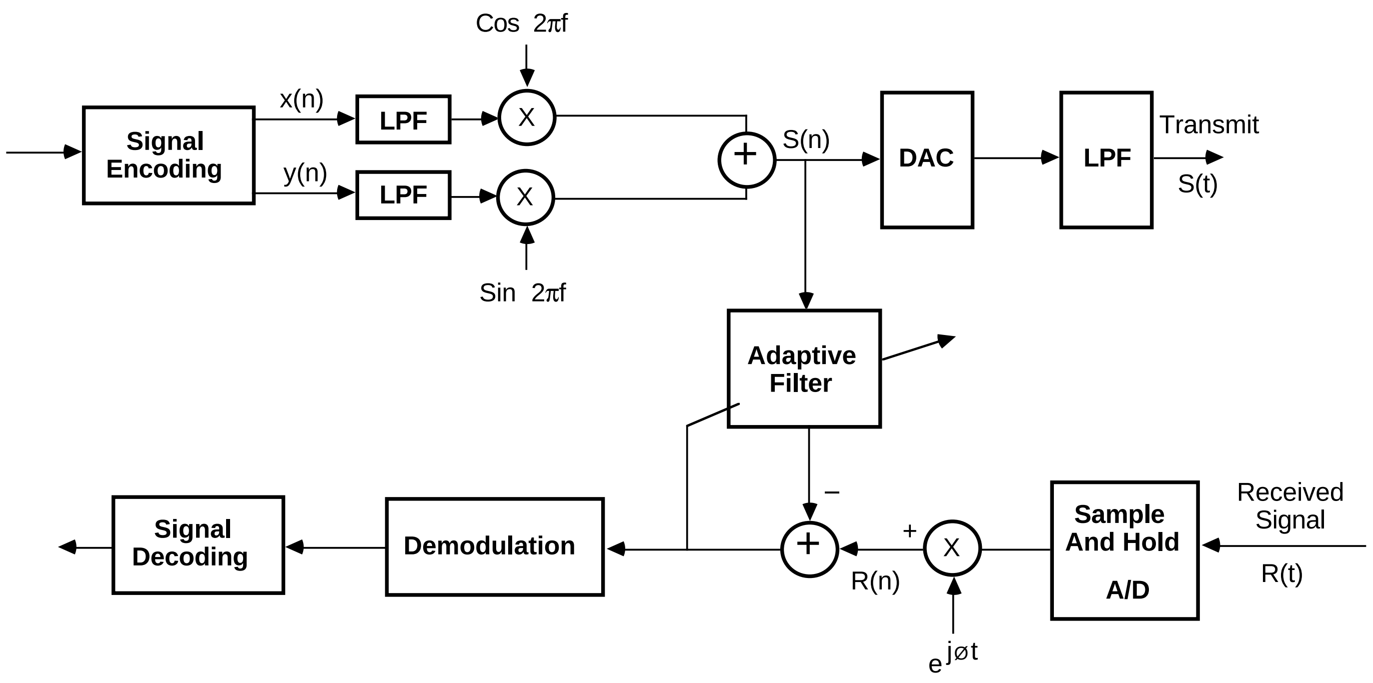

Figure 2.25 V.32 Modem Block Diagram

The CCITT specification for V.32 modems recommends a carrier frequency of 1800±7 Hz. The echo canceller must be able to cancel 16 ms of echo. At 9600 samples / second, a 154-tap FIR filter is required to cancel the echo. It is recommended that the echo canceller be implemented with a minimum number of taps.

Assuming that the canceller and frequency shifter have converged during the training period, about 200 cycles are required to cancel a V.32 signal. Benchmarks are summarized in Table 2.4.

Operation	*Cycles*	*@12.5 MHz*
Real FIR Filter	N + 6	80 ns per tap
Complex FIR Filter	4 (N–1) + 21	240 ns per tap
Real LMS Update (Stochastic)	2N + 9	160 ns per tap
Complex LMS Update (Stochastic)	6N + 10	480 ns per tap
154-Tap LMS Filter With Update	935	74.8 µs

N = Number of Taps

Table 2.4 ADSP-2100 Family Benchmarks For Echo Cancellation