2.5 ADAPTIVE EQUALIZATION

This section presents subroutines for an ADSP-2100 family implementation of an adaptive channel equalizer for a high speed modem. The CCITT's V.32 recommendation for a 9600 bps modem specifies the use of this type of equalizer in the receiver section.

The architecture used in this equalizer is a fractionally-spaced tapped delay line with a least-mean-squared (LMS) algorithm for adapting the tap weights.

The topics discussed in this section are:

- Historical perspective of adaptive filters
- Applications of adaptive filters
- Channel equalization in a modem
- Equalizer structures
- Least Mean Square (LMS) Algorithm
- Program Structure
- Practical considerations

2.5.1 History Of Adaptive Filters

Until the mid-1960s, telephone-channel equalizers were either fixed equalizers that caused performance degradation or manually adjustable equalizers that were cumbersome to adjust.

In 1965, Lucky (see "References" at the end of this chapter) introduced the zero-forcing algorithm for automatic adjustment of the equalizer tap weights. This algorithm minimizes a certain distortion, which has the effect of forcing the intersymbol interference (ISI) to zero. This breakthrough by Lucky inspired other researchers to investigate different aspects of the adaptive equalization problem, leading to new improved solutions.

Proakis and Miller (1969) reformulated the adaptive equalizer problem using a new criterion known as the mean squared error (MSE). This formulation requires a relatively modest amount of computation and remains the most popular approach for data rates up to 9600 bits/s.

Three years later, Ungerboeck (1972) improved on this work by presenting a detailed mathematical analysis of the convergence properties of an adaptive transversal equalizer using the least-mean-squared (LMS) algorithm. This algorithm is described later in this chapter.

A more powerful algorithm for adjusting the tap weights based on Kalman filtering theory was developed soon afterward (Godard, 1974). This algorithm is computationally demanding, but it was later modified by Falcomer and Ljing (1978) to simplify its computational complexity.

All of these adaptive equalizer implementations are synchronous, that is, the spacing between taps is equal to the reciprocal of the symbol interval. Other possible structures include the fractionally spaced equalizer (FSE) and the decision feedback equalizer (DFE).

The FSE has the ability to better compensate for channel distortion by spacing the tap weights more closely than in the conventional synchronous equalizer. Brady (1970) did some early work on this class of equalizers and was followed by Ungerboeck (1976). The DFE, on the other hand, uses a more elaborate structure and can yield good performance in the presence of severe ISI as experienced in fading radio channels.

2.5.2 Applications Of Adaptive Filters

Adaptive filters offer a significant improvement in performance over fixed-tap-weight digital filters because of their ability to detect signals in environments of unknown characteristics. They are successfully used in several areas including:

System Identification And Modeling

An adaptive transversal filter can be forced to converge to the same impulse response as an unknown linear system and then can be used to model the unknown system. To determine the taps for this filter, an excitation input drives both the unknown system and the adaptive filter. The outputs of these two systems are compared, and the error signal generated is used to adjust the tap weights of the adaptive filter to reduce the error size. After a sufficiently large number of iterations, the error is reduced to some small value (in a statistical sense) and the tap weights converge to model the real system.

If the unknown system is dynamic and time-variant, the adaptive filter can track these variations provided they are sufficiently slow compared to the convergence time of the filter.

Echo Cancellation
In telephone systems that include both 2-wire and 4-wire loops, hybrid circuits couple these lines. These hybrid circuits create impedance mismatches which in turn create signal reflections, heard at both ends of the line as echo. This echo is tolerable to some degree over long distance voice connections, but can be catastrophic in high-speed data transmission over cross-Atlantic links.

Echo cancellers, in the form of adaptive filters, model the impulse response of the echo path. Cancellation is achieved by making an estimate of the echo and subtracting it from the return signal.

Linear Predictive Coding
In the past 20 years, digital coding of speech waveforms has become a popular technique for reducing speech degradation due to transmission. Of the speech coding techniques, linear predictive coding (LPC) stands out for its ability to produce low data rates. Basic speech parameters (e.g. pitch, vocal tract, formants) are estimated, transmitted and then used at the receiver to resynthesize the speech through a speech production model. Adaptive filters can be used to estimate speech parameters in model-based speech coding systems.

The speech quality of LPC is synthetic when compared to other coding techniques such as PCM or ADPCM; however, its significantly lower data rates make it attractive. The GSM standard for the Pan-European cellular digital mobile radio network specifies an LPC-based coding scheme.

Adaptive Beamforming
A spatial form of adaptive signal processing finds applications in radar and sonar. By combining signals from an array of sensors, it is possible to change the directivity pattern of the array. Independent sensors (e.g. antennas or hydrophones) placed at various locations in space or water detect incoming waveforms. The collection of sensor outputs at a particular instant is analogous to the set of consecutive tap inputs in a transversal filter. The sensitivity and directivity of the sensor array can be adaptively adjusted. Beamforming is discussed in Chapter 15 of *Digital Signal Processing Using the ADSP-2100 Family*.

Adaptive Channel Equalization For Data Transmission
Adaptive filters used in digital communication systems as channel equalizers minimize transmission distortion and maximize the use of channel bandwidth. A typical bandlimited telephone channel or radio link suffers from intersymbol interference (ISI) and additive noise. To improve system performance in additive-noise channels, transmission power can be increased. However, increased power has no effect on ISI since it amplifies both the intended symbol sample as well as interfering ones.

The traditional technique for alleviating ISI is an equalizing filter at the receiver. The receiver equalizer filter combines the channel characteristics and the transmitter filter to minimize ISI distortions. Channel characteristics, however, vary over time. An adaptive equalizer is needed to ensure a constant transmission quality.

Since the channel conditions are unknown, a training sequence is transmitted to bring up the equalizer from its initial (usually zero) state. This sequence is known at the receiver and therefore the deviation error of received samples from the expected sequence is used to adjust the equalizer tap weights. Once the training period is completed, the weights can still be continually updated in a decision directed mode. In this mode, a minimum distance detector at the receiver decides which symbol was transmitted. In normal operation these decisions have a high probability of being correct, and thus are good enough to allow the equalizer to maintain proper adjustment.

2.5.3 Channel Equalization In A Modem

The International Telegraph and Telephone Consultative Committee (CCITT) sets standards and protocols for telephone and telegraph equipment. Its V.32 modem recommendation specifies a fractionally spaced transversal filter as the channel equalizer in the receiver. This equalizer, along with trellis coding and quadrature amplitude modulation (QAM), maximizes data rates over the bandlimited telephone channel.

A telephone channel can suffer from a variety of limitations as a communications medium:

- As a bandlimited channel, it creates an environment for ISI.
- Channel additive noise requires increased transmitted power to improve signal-to-noise ratio.
- Radio links create fading channels and echo in cross-Atlantic connections
- When several connections are frequency multiplexed, baseband speech signals are modulated into the passband using different carrier frequencies for transmission. Demodulating these passband signals can create frequency offsets as well as amplitude and phase distortion.
- Phase jitter (poor timing recovery).
- Envelope delay or harmonic distortion is another limitation.

These channel limitations combined with the dense symbol constellation of the V.32 modem necessitate adaptive equalization for acceptable error rates at 9600 bits / s.

2.5.3.1 Equalization

The basic function of the equalizer is to create an ideal transmission medium from a real channel. An example channel's short impulse response {h1, h2, h3, h4} is shown in Figure 2.26. The ideal medium is characterized as a pure delay, shown in Figure 2.27.

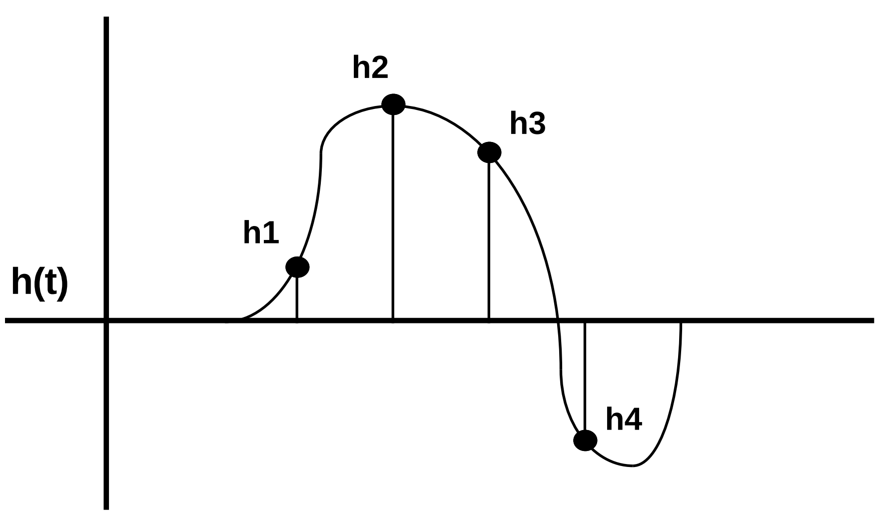


Figure 2.26 Example Short Impulse Response

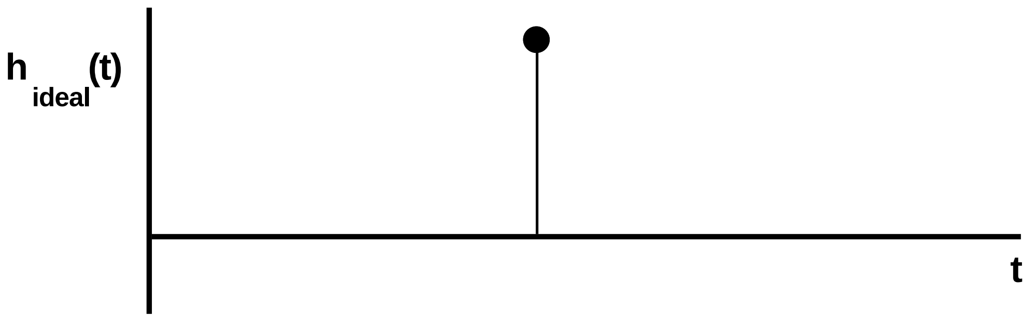


Figure 2.27 Pure Delay Impulse Response

Take for example the equalizer shown in Figure 2.28 which has three taps {c1, c2, c3}. Convolving this response with the channel's impulse response from Figure 2.26 yields

$$\begin{vmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{vmatrix} = \begin{vmatrix} c_1 & 0 & 0 & 0 \\ c_2 & c_1 & 0 & 0 \\ c_3 & c_2 & c_1 & 0 \\ 0 & c_3 & c_2 & c_1 \\ 0 & 0 & c_3 & c_2 \\ 0 & 0 & 0 & c_3 \end{vmatrix} \times \begin{vmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{vmatrix}$$

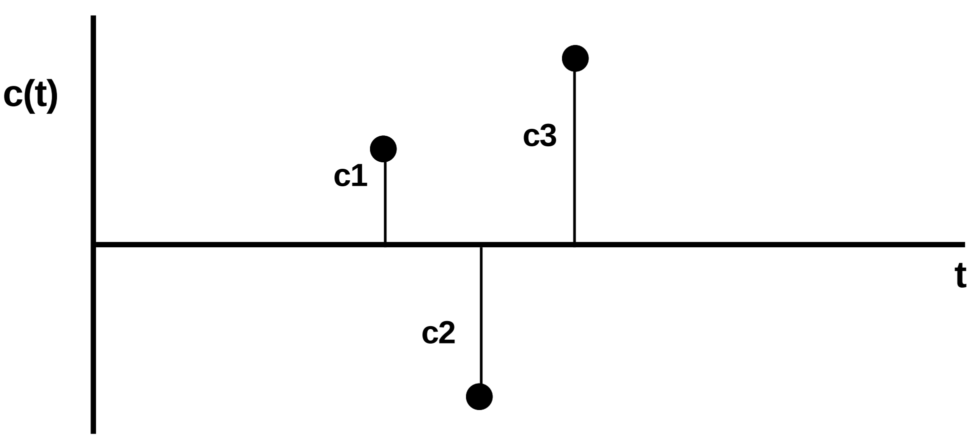


Figure 2.28 Equalizer Impulse Response

The outputs $\{y_1, y_2, y_3, y_4, y_5, y_6\}$ represent samples of the impulse response of the combined channel/equalizer system.

If the equalizer is to create ideal conditions for transmission, all the y's should be zeros except for one main sample. Rewriting the equation for ideal equalization yields:

$$\begin{vmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{vmatrix} = \begin{vmatrix} c_1 & 0 & 0 & 0 \\ c_2 & c_1 & 0 & 0 \\ c_3 & c_2 & c_1 & 0 \\ 0 & c_3 & c_2 & c_1 \\ 0 & 0 & c_3 & c_2 \\ 0 & 0 & 0 & c_3 \end{vmatrix} \times \begin{vmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{vmatrix}$$

or

$$\begin{aligned} 0 &= c_1h_1 \\ 0 &= c_1h_2 + c_2h_1 \\ 1 &= c_1h_3 + c_2h_2 + c_3h_1 \\ 0 &= c_1h_4 + c_2h_3 + c_3h_2 \\ 0 &= c_2h_4 + c_3h_3 \\ 0 &= c_3h_4 \end{aligned}$$

The system of equations above has only three controllable variables (unknowns) but six simultaneous equations. The system is overdetermined and can only be solved approximately. To approximate this solution, a reformulation of a recursive technique known as method of steepest descent can be used. This iterative algorithm is defined by the equation:

(1) $C_{k+1} = C_k - \Delta\, \partial E / \partial C_k$

where E is a defined performance index to be optimized. It is a function of some controllable parameters (tap weights C_k). E is minimized by adjusting the tap weights in small steps (Δ). The gradient vector $\partial E / \partial C_k$ indicates the direction of the adjustment required to minimize E. This method converges to an optimum solution when $\partial E / \partial C_k$ is zero.

2.5.3.2 Performance Index

It is important to choose a meaningful performance index that is a linear function of the tap weights and that defines a smooth error surface (bowl) in the space spanned by the tap weight vector. This ensures the convergence of the algorithm to the lowest point (minimum) of the error surface.

In some cases, a desirable performance index is a nonlinear function of the adjustable parameters and the solution is unrealizable. As an example, consider the probability of error in a digital communication system. Even though this is a meaningful measure of system performance, it is a highly nonlinear function of the equalizer tap weights. Using the method of steepest descent, it cannot be determined whether the adaptive equalizer has converged to the optimum solution or to one of the relative minima of the surface. For this reason some desirable performance indices must be rejected.

A practical and popular index for performance is the mean squared error (MSE). The error is measured as the difference between the received signal and the ideal signal value. The MSE index is a measure of the energy in this error signal averaged over a signaling interval. It results in a quadratic performance surface as a function of the filter coefficients and thus has a single minimum (optimal solution). An implementation of an MSE-based iterative adaptation algorithm is developed for the ADSP-2100 processor family in this chapter; it is discussed in a later section.

2.5.4 Equalizer Architectures

The preferred form of a linear equalizer is a tapped delay line. The delay line consists of delay elements in a feedforward path and possibly a feedback path.

If the delay line has feedforward delays only, its transfer function can be expressed as a single polynomial in Z^{-1} and therefore the equalizer has a finite impulse response (FIR). This type of equalizer is often called a nonrecursive or transversal equalizer (Figure 2.29).

If the delay line also has feedback delay elements, its transfer function is a rational function of Z^{-1} and the equalizer has an infinite impulse response (IIR) due to its nonzero poles (Figure 2.30).

The V.32 modem equalizer has no feedback delay elements and is therefore an FIR equalizer.

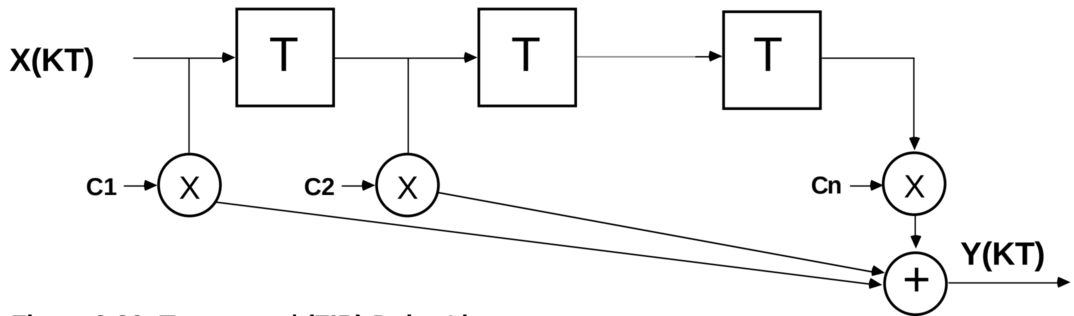


Figure 2.29 Transversal (FIR) Delay Line

2.5.4.1 Real Or Complex

In a one-dimensional communication system (e.g. pulse amplitude modulation or PAM), the signal is real and the equalizer has real coefficients. The V.32 modem, which uses quadrature amplitude modulation (QAM), transmits complex data by modulating two

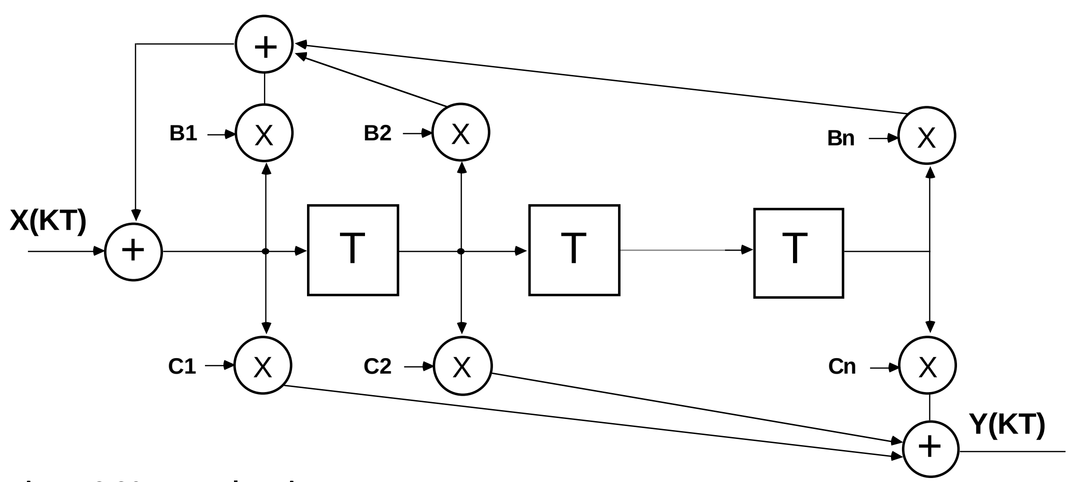


Figure 2.30 IIR Delay Line

orthogonal carrier signals. Because of cross-distortion between the in-phase and quadrature channels in this two-dimensional communication system, an equalizer with complex tap coefficients is required.

Algorithms for the complex equalizer are essentially the same as for the real equalizer with the added burden of complex arithmetic. A complex equalizer typically requires four times as many multiplications and introduces the complex conjugation operator in recursive algorithms such as LMS adaptation.

2.5.4.2 Sampling Rates

It is often advantageous to space the delay elements in an equalizer more closely than the symbol rate, as shown in Figure 2.31. This has the effect of oversampling the input to the filter and thus increasing the effective bandwidth of the equalizer. The input is pushed onto the delay line twice for every one output computed. Fractionally spaced equalizers have superior performance because of wider bandwidth, and they simplify the problem of phase synchronization between transmitter and receiver. They do, however, suffer from stability problems in low noise conditions and are more computationally demanding (Ungerboeck, 1976).

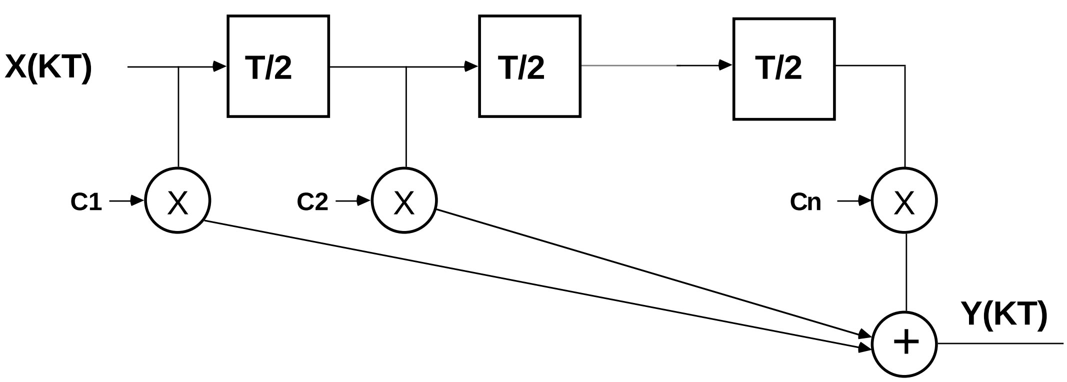


Figure 2.31 Fractionally Spaced Delay Line (FSE)

A fractionally spaced filter can be designed the same way as a T-spaced delay line filter. The basic delay line structure is the same for both. For a T/2 FSE filter, the samples are shifted in at $2f_s$ (twice the sampling frequency) but the output is only computed at f_s, i.e. every other input time.

The ADSP-2100 routine to implement the delay line with complex tap weights is in Listing 2.13.

```
{        Fractionally Spaced Filter (FSE) Subroutine

This Complex Fractionally Spaced Filter (FSE) Subroutine is used in the V32
equalizer. The basic structure for the delay line is the same as that of a
T-Spaced Filter (TSE). In the FSE case, however, samples are shifted in at
2Fs (Fs=Sampling Frequency) and the output is computed at Fs, i.e. at
alternate times. This subroutine will therefore be called after 2 new input
samples have been pushed onto the delay line.

         Calling Parameters
            I0—>Oldest data value in real delay line (Xr’s)
            L0=filter length (N)
            I1—>Oldest data value in imag. delay line (Xi’s)
            L1=filter length (N)
            I4—>Beginning of real coefficient table (Cr’s)
            L4=filter length (N)
            I5—>Beginning of imaginary coefficient table (Ci’s)
            L5=filter length (N)
            M0,M6=1
            AX0=filter length minus one (N-1)
            CNTR=filter length minus one (N-1)

         Return Values
            I0—>Oldest data value in real delay line
            I1—>Oldest data value in imaginary delay line
            I4—>Beginning of real coefficient table
            I5—>Beginning of imaginary coefficient table
            SR1=real output (rounded, cond. saturated)
            MR1=imaginary output (rounded, cond. saturated)

         Altered Registers
            MX0,MY0,MR,SR1

         Computation Time
            2*(N-1)+2*(N-1)+13+8 cycles

All coefficients and data values are assumed to be in 1.15 format.
}
```

```
fir:      MR=0, MX0=DM(I1,M0), MY0=PM(I5,M6);
          DO realloop UNTIL CE;
             MR=MR-MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M6);  {Xi*Ci}
realloop:    MR=MR+MX0*MY0(SS), MX0=DM(I1,M0), MY0=PM(I5,M6);  {Xr*Cr}
          MR=MR-MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M6);     {last Xi*Ci}
          MR=MR+MX0*MY0(RND);                                  {last Xr*Cr}
          IF MV SAT MR;
          SR1=MR1;                                             {Store Yr}
          MR=0, MX0=DM(I0,M0), MY0=PM(I5,M6);
          CNTR=AX0;
          DO imagloop UNTIL CE;
             MR=MR+MX0*MY0(SS), MX0=DM(I1,M0), MY0=PM(I4,M6);  {Xr*Ci}
imagloop:    MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I5,M6);  {Xi*Cr}
          MR=MR+MX0*MY0(SS), MX0=DM(I1,M0), MY0=PM(I4,M6);     {last Xr*Ci}
          MR=MR+MX0*MY0(RND);                                  {last Xi*Cr}
          IF MV SAT MR;                                        {MR1=Yi}
          RTS;
```

Listing 2.13 Delay Line Routine, Complex Tap Weights

2.5.5 Least Mean Squared (LMS) Algorithm

Since the mean squared error (MSE) performance index is a convex function of the tap weights (has a bowl-shaped surface), the optimum tap weights can be obtained by the steepest descent algorithm. In this algorithm, tap weights are assumed to have an arbitrary initial setup and are moved in the direction of optimum value when MSE is minimized. The direction is determined by the gradient of the objective function of performance,

(2) $$E = |\, e(kt) \,|^2$$

where e(kt) is the error between the estimated symbol and the received sample and the bar above the expression denotes time averaging. Optimum tap weights are determined when the derivative of the MSE surface with respect to all the tap weights is zero.

(3) $$\partial E / \partial C_k = 0 \qquad 1 \le k \le N, \text{ for an N-tap filter}$$

The error function E is a complex quadratic function because of the 2-dimensional modulation scheme (QAM). The derivative expression is:

(4) $$\partial E / \partial C_n(k) = -2\, \overline{e(kt)\ y(kT_{sym} - nT_{taps})}$$

where T_{taps} is the spacing between the taps
T_{sym} is the spacing between symbols

Combining with equation (1) yields:

$$C_n(k+1) = C_n(k) + \beta\overline{e(kt)\, y^*(kT_{sym} - nT_{taps})} \tag{5}$$

The implementation of the steepest descent algorithm requires the evaluation of the cross-correlation of error signal e(kt) and received signal y(t). Cross-correlation requires time-averaging, which is not a viable option considering the real time requirements of the equalizer. To alleviate this problem, the approximation:

$$\overline{e(kt)\, y^*(kT_{sym} - nT_{taps})} \approx e(kt)\, y^*(kT_{sym} - nT_{taps}) \tag{6}$$

is used instead of time-averaging. This simplification of the steepest descent algorithm greatly reduces the amount of computation. It is very popular and is generally referred to as the least mean square (LMS) algorithm.

An LMS algorithm updates the equalizer tap weights according to

$$C_n(k+1) = C_n(k) + \beta e(kt)\, y^*(kT_{sym} - nT_{taps}) \tag{7}$$

Listing 2.14 shows an LMS algorithm implemented on the ADSP-2100 family.

```
{        Complex SG Update LMS Subroutine.

This routine updates the complex taps according to the relation:

         Cn(k+1)=Cn(k)-Beta.E(k).Y*(n-K)

         where:      <Beta>=Adaptation step size
            <E(k)>=estimation error at time k
            <Y*(n-k)>=Received signal complex
                     conjugated & sampled at time (n-k)

         Calling Parameters
            I0—>Oldest data value in real delay line   L0=N
            I1—>Oldest data value in imag. delay line  L1=N
            I4—>Beginning of real coefficient table    L4=N
            I5—>Beginning of imag coefficient table    L5=N
            MX0=real part of Beta*Error
            MX1=imag part of Beta*Error
            M0,M5=1
            M1=-1
            M6=0
            CNTR=Filter length (N)

         Return Values
            Coefficients updated
            I0—>Oldest data value in real delay line
            I1—>Oldest data value in imag delay line
            I4—>Beginning of real coefficient table
            I5—>Beginning of imag coefficient table

         Altered Registers
            MY0,MY1,MR,SR,AY0,AY1,AR

         Computation Time
            6*N+10 cycles

         All coefficients and data values are assumed to be in
         1.15 format.
}

upd_taps:   MY0=DM(I0,M0);                            {Get Xr}
            MR=MX0*MY0(SS), MY1=DM(I1,M0);            {Er*Xr, get Xi}
            DO adaptc UNTIL CE;
               MR=MR+MX1*MY1(RND), AY0=PM(I4,M6);     {Ei*Xi, get Cr}
               AR=AY0-MR1, AY1=PM(I5,M6);             {Cr-(Er*Xr+Ei*Xi), get Ci}
               PM(I4,M5)=AR, MR=MX1*MY0(SS);          {Store new Cr, Ei*Xr}
               MR=MR-MX0*MY1(RND), MY0=DM(I0,M0);     {Er*Xi, get Xr}
               AR=AY1-MR1,MY1=DM(I1,M0);              {Ci-(Ei*Xr-Er*Xi), get Xi}
adaptc:     PM(I5,M5)=AR, MR=MX0*MY0(SS);             {Store new Ci, Er*Xr}
            MODIFY (I0,M1);                           {point back to start}
            MODIFY (I1,M1);                           {of complex delay line}
            RTS;
```

Listing 2.14 LMS Routine

2.5.6 Program Structure

The flowchart shown in Figure 2.32 depicts the sequence of operations of an equalizer program. Each program section is discussed below.

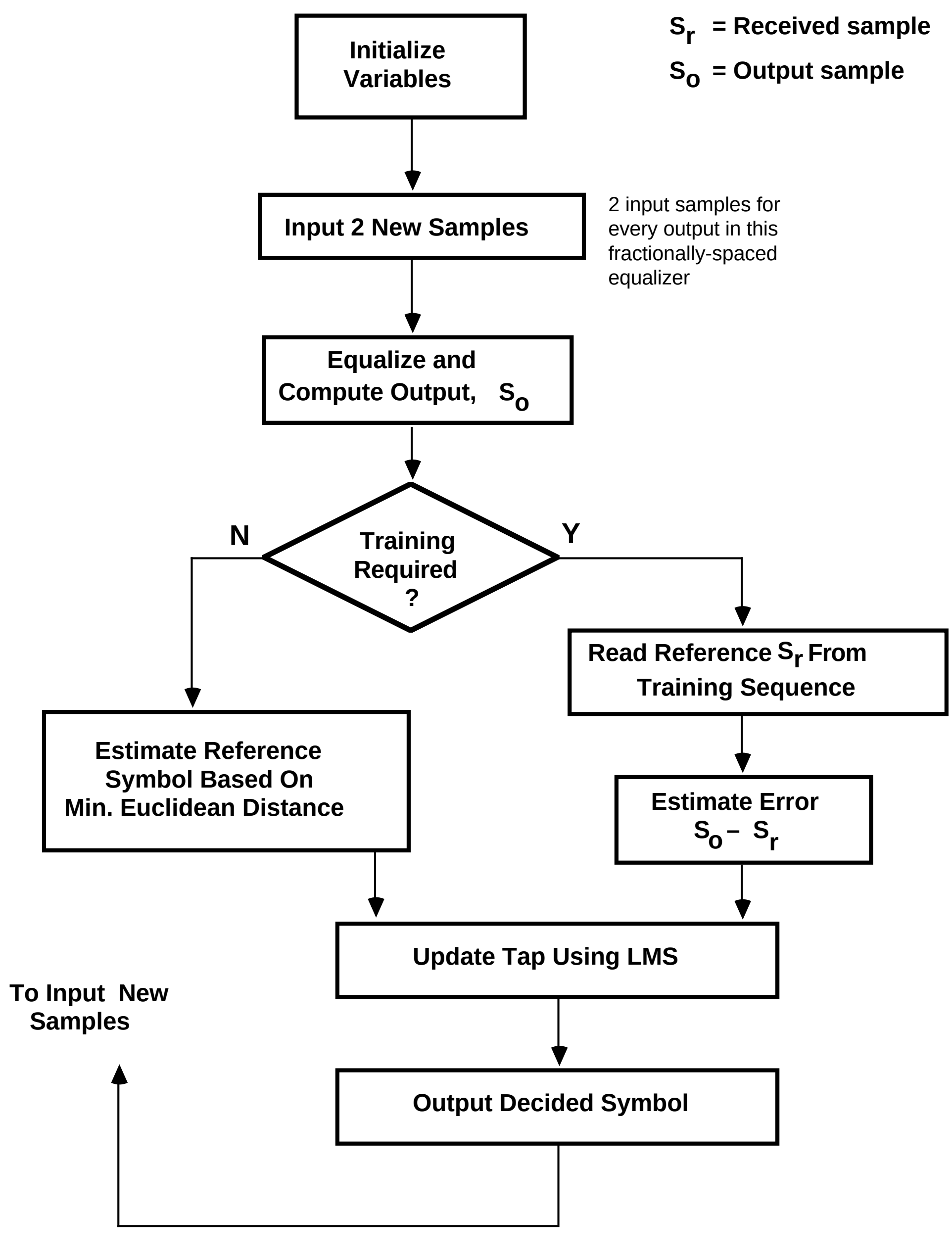


Figure 2.32 Adaptive Equalizer Flowchart

2.5.6.1 Input New Sample

The equalizer program is interrupt-driven. The arrival of a new complex sample causes the equalizer to start executing. The *sample_in* port in Listing 2.15 holds the new sample (real, then imaginary). Index registers I0 and I1 point to the complex delay line.

The V.32 modem recommendation specifies a fractionally spaced equalizer. The delay line therefore consists of delays that are spaced at one-half the symbol rate. This means that the output (at 2400 symbols/s) is only computed for every two input samples (at 4800 symbols/s). The variable *decimator_flag* is used to decide whether to get another sample or to start computing the output.

```
{        input_new_sample routine

This part will read a new sample from the port ‘sample_in’ and
place it on the delay line. This new complex sample will
overwrite the oldest value on the delay line (complex also).
}

start:   AR=DM(sample_in);          {read in real & imag. values}
         DM(I0,M0)=AR;              {of new sample and store
them}
         AR=DM(sample_in);          {in delay line}
         DM(I1,M0)=AR;
         AR=DM(decimator_flag);  {check flag to see if filtering}
                                 {is required this time through.}
         AR=NOT AR;                 {Then toggle the flag}
         DM(decimator_flag)=AR;     {to ensure that we filter}
                                    {every other sample}
         IF EQ RTS;                 {as required in an FSE}
```

Listing 2.15 Input Routine

2.5.6.2 Filtering (Equalizing)

The actual filtering is performed in the subroutine in Listing 2.16. The calling parameters for the filter are initialized, and after the subroutine is called the return values are stored in data memory.

```
{        do the fir filtering (equalization)

Performs the actual fir filtering. Takes the input sample
from the receiver front end & produces an output value
(fir_out_real & fir_out_imag)
}
         AX0=no_of_taps-1;
         CNTR=no_of_taps-1;
         CALL fir;
         DM(fir_out_real)=SR1;       {save return values of}
                                     {subroutine in}
         DM(fir_out_imag)=MR1;       {their designated var names:}
                                     {fir_out_real & fir_out_imag}
```

Listing 2.16 Filter Routine

2.5.6.3 Training Sequence

Initially the tap weights of the equalizer are at some arbitrary state (possibly zero) that is typically far from the optimum state. The receiver decisions based on the output of the equalizer are therefore incorrect with a high probability. Decision-directed adaptation is not guaranteed to work because of the initial high error rate. The equalizer might be unable to move into the error-free region and the adaptation would diverge or stop (MSE neither increasing nor decreasing significantly).

To train the equalizer through this blind stage, a data sequence that is known at the receiver is used for initial transmission. If the locally generated reference is properly synchronized to the received signal, this training brings the equalizer to its optimum state. After training, slow channel variations are tracked using decision-directed adaptation.

The stored training sequence at the receiver is read at the *training_list* port (real, then imaginary) in Listing 2.17. The received signal is read in from the filter outputs *fir_out_real* and *fir_out_imag*. A complex error value which is equal to the Euclidean distance between the two samples is generated. The estimation error is stored in data memory (*error_real* and *error_imag*).

```
{       estimate the transmitted symbol ( training )

Given fir_out_real & fir_out_imag, we compute the error value
(real and complex) using the training sequence as a reference.
This estimate for error is used only initially to train the
equalizer. Following the training, decision directed adaptation
would take over.
}

         AX0=DM(fir_out_real);      {inputs are fed in directly}
         AX1=DM(fir_out_imag);      {from output of fir}
         AY0=DM(training_list);
         AY1=DM(training_list);
         CALL est_error_train;

{————————————————————————————————}
{
Est_error_train subroutine: Returns the equalizer output minus
the
ideal value available from the training sequence.

         AX0=fir_out_real
         AX1=fir_out_imag
         AY0=ideal_symbol_real
         AY1=ideal_symbol_imag

         Returns:
            error_real
            error_imag

}

est_error_train:  AR=AX0-AY0;
                  DM(error_real)=AR;
                  AR=AX1-AY1;
                  DM(error_imag)=AR;
                  RTS;
```

Listing 2.17 Training Sequence Routine

2.5.6.4 Decision-Directed Adaptation

Once the equalizer is trained, decision-directed adaptation is possible. In this mode, symbols estimated at the receiver are used as the reference from which to measure the deviation error and subsequently adjust the taps. With the equalizer trained, low decision-error rates make it possible to continue to adapt to small changes in channel conditions.

In Listing 2.18, the estimated symbol is chosen as the symbol geometrically closest to the received coordinates. A 15-instruction loop (worst case) computes the distance to each of the 32 symbols in the symbol table and determines the nearest one. The routine returns a pointer to the estimated symbol in the table as well as the real and imaginary values of the error.

```
{              estimate the transmitted symbol    ( no training )

Given fir_out_real & fir_out_imag, we compute the error value (real and
complex) using a Euclidean distance routine (decision directed adaptation).
In this mode the estimated symbol is the geometrically closest to the
received coordinates.  This routine also returns the complex error value.
}
        AX0=DM(fir_out_real);   {these inputs are fed in directly}
        AX1=DM(fir_out_imag);   {from the output of the fir}
        CALL est_error_eucl;
{————————————————————————————————}
{Estimate_error_euclidean Symbol Subroutine
(normal mode, i.e. no training):

Maps input sample onto an ideal symbol in the constellation table This
routine also returns the value of the error measured as the Euclidean
distance between received signal and its ideal value.

         Calling Parameters
            AX0 contains Xr
            AX1 contains Xi
            M0=1
            M1=-1

         Return Values
            SI=decision index j
            (position with respect to end of table)
            AF=minimum distance (squared)
            I2—>Beginning of constellation table

         Altered Registers
            AY0,AY1,AF,AR,MX0,MY0,MY1,MR,SI
            AR_SAT mode enabled

         Computation Time
            15*N+5 (maximum)
}
```

```
est_error_eucl:   I2=^constellation_table;
                  L2=3;                      {number of symbols in}
                                             {constellation table}
                  AY0=32767;                 {Initialize minimum distance to}
                                             {largest possible value}
                  ENA AR_SAT;                {put ALU in saturation mode to}
                                             {prevent overflow}
                  AF=PASS AY0, AY0=DM(I2,M0);         {Get Cr}
                  CNTR=32;
                  DO ptloop UNTIL CE;
                     AR=AX0-AY0, AY1=DM(I2,M0);       {Xr-Cr, Get Ci}
                     MY0=AR, AR=AX1-AY1;              {Copy Xr-Cr, Xi-Ci}
                     MY1=AR;                          {Copy Xi-Ci}
                     MR=AR*MY1(SS), MX0=MY0;          {(Xi-Ci)^2,}
                                                      {Copy Xr-Cr}
                     MR=MR+MX0*MY0(SS);               {(Xr-Cr)^2}
                     IF MV SAT MR;           {clip result to max value}
                     AR=MR1-AF;              {Compare with previous minimum}
                     IF GE JUMP ptloop;
                     AF=PASS MR1;            {New minimum if MR1<AF}
                     AR=AX0-AY0;             {error is euclidean distance}
                     DM(error_real)=AR;      {between actual received}
                     AR=AX1-AY1;             {signal and ideal symbol}
                     DM(error_imag)=AR;      {coordinates}
                     SI=CNTR;                {Record constellation index}
ptloop:              AY0=DM(I2,M0);
                  MODIFY(I2,M1);             {Point to beginning of table}
                  RTS;
```

Listing 2.18 Decision-Directed Adaptation Routine

2.5.6.5 Tap Update (LMS Algorithm)

Once an estimate error is computed, it is possible to adapt the equalizer coefficients to a new set of values closer to the optimum vector. The LMS routine in Listing 2.19 performs the computation. The estimation error is first scaled down by the adaptation step size (β). This constant provides a mechanism to trade off convergence speed against the amount of jitter in the steady state value of the tap vector.

```
{        update the taps

Takes the estimation error values previously computed multiply
them by the step size (beta). The upd_taps routine is then called
to update coefficients of the equalizer.

}


         MY0=DM(error_real);  {MX0=beta x error_real}
         MX0=beta;
         MR=MX0*MY0(SS);
         MX0=MR1;

         MY1=DM(error_imag);
         MX1=beta;            {MX1=beta x error_imag}
         MR=MX1*MY1(SS);
         MX1=MR1;
         CNTR=No_of_taps;
         CALL upd_taps;
```

Listing 2.19 Tap Update Routine

2.5.6.6 Output

These equalizer routines can be integrated into other modules to form the receiver block of a V.32 modem. As specified in the V.32 recommendation, the equalized sample is decoded using the Viterbi algorithm. The equalizer output (real and the imaginary) is therefore written to an I/O port *sample_out.*

```
{        output the resulting sample of the equalizer}

         AR=DM(fir_out_real); {output the equalizer output}
         DM(sample_out)=AR;   {to the outport port}
         AR=DM(fir_out_imag);
         DM(sample_out)=AR;
         RTS;                 {return from equalizer routine and}
                              {wait for a new sample interrupt}
```

Listing 2.20 Output Routine

2.5.7 Practical Considerations

This section describes considerations for using and modifying the routines in this chapter.

2.5.7.1 Viterbi Decoder

In the implementation of decision-directed adaptation, the received sample is matched to the nearest symbol and the error is used to adjust the taps. A few wrong decisions could cause the equalizer to wander off temporarily, but because right decisions have a proportionately larger effect, convergence is ensured.

If a sophisticated algorithm such as Viterbi decoding is used to improve the decision, the signal sample and error are not available until several symbol intervals after the input time. This Viterbi delay requires a modified LMS updating routing with delayed coefficient adaptation (DLMS). It can be shown that the DLMS adaptation has the same steady state behavior as the LMS adaptation, provided the adaptation constant is within a certain range (Long et al, 1989).

2.5.7.2 Pseudo-Random Training Sequence

The routines in this chapter have been validated with a pseudo-random training sequence. This training sequence consists of a set of symbol values with a repetition period that is much longer than the convergence time of the equalizer. The benefit of using such a sequence is that the approximation of the gradient vector $\partial E/\partial C_k$ is less noisy. Noisy estimates of the gradient vector can cause the tap coefficients to wander a long way from the path of the steepest descent (Bingham, 1988).

2.5.7.3 Delay Line Length

If the exact source of the channel's distortion is known and the impulse response can be modeled precisely, it is possible to calculate the minimum order of the equalizer transfer function needed to reduce the MSE to an acceptable level. In general, the only practical method of deciding the length of the delay line is to derive a theoretical length based on several worst-case channel characteristics. The equalizer is then designed slightly longer than the theoretical minimum to compensate for the cumulative effects of finite precision arithmetic in the ADSP-2100 family processor. For a discussion of quantization effects in the LMS algorithm, see Bershad, 1989.

2.6 CONTINUOUS PHASE FREQUENCY-SHIFT KEYED MODULATION

Constant phase modulation (CPM) techniques find applications in satellite communications. Because of power amplifier considerations, satellite communications require a modulation technique with a constant or nearly constant envelope versus time (no amplitude modulation). Technological and regulatory limitations also require low error probability for a given signal-to-noise ratio and high bits per second of transmitted information for a given bandwidth. The technique of multi-*h* CPM, which combines encoding and modulation, achieves all of these goals.

This chapter describes an implementation of continuous phase frequency-shift keying (CPFSK), a sub-class of multi-*h* CPM, on the ADSP-2100 family of processors. Only modulation is described here; demodulation is usually performed with the Viterbi algorithm.

Fast frequency-shift keying (FFSK) is a special case of CPFSK with h=1/2.

2.6.1 CPFSK Methodology

The general form for a multi-*h* CPM signal is:

$$s(t;\alpha) = \sqrt{(2E_s / T_s)} \cos\left[2\pi f_0 t + \varphi(t;\alpha) + \varphi_0\right]$$

E_s = symbol energy
T_s = symbol duration
f_0 = carrier frequency (Hz)
φ_0 = carrier phase (arbitrary)
$\varphi(t;\alpha)$ = information-carrying phase function, expressed as:

$$2\pi \int_{-\infty}^{t} \sum_{i=-\infty}^{\infty} h_i a_i \, g(\tau - iT_s)\, d\tau \qquad -\infty < t < \infty$$

where:

α = $(\ldots, a_{-2}, a_{-1}, a_0, a_1, a_2, \ldots)$, representing the data sequence
h_i = set of K modulation indices cycled through periodically, i.e., $h_{i+K} = h_i$
$g(t)$ = frequency pulse-shape function

For CPFSK, all h_i are equal and the pulse-shape function is:

$$g(t) = T_s / 2 \qquad \text{for } 0 \le t \le T_s\text{, otherwise } 0$$

2.6.2 ADSP-2100 Family Implementation

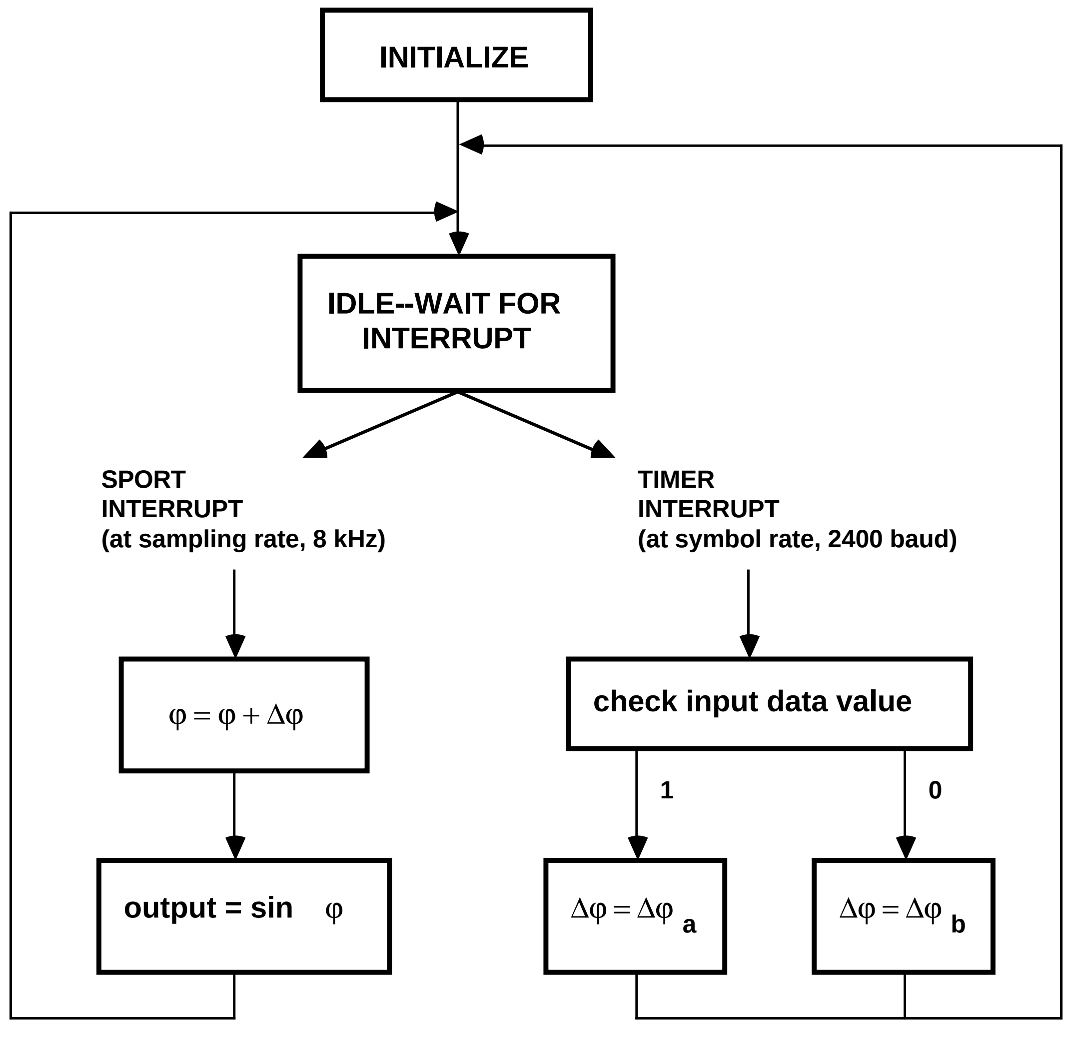
Figure 2.33 shows a flowchart of the CPFSK program implemented on the ADSP-2100 family of processors. This particular implementation uses the ADSP-2101 to take advantage of its on-chip serial ports and timer. The timer generates a clock at the symbol rate (2400 baud) for reading input data. The ADSP-2101 outputs CPFSK modulated data to a digital-to-analog converter (DAC) at the rate of 8 kHz.

The CPFSK program is shown in Listing 2.21. This program sets up a buffer of dummy data for demonstrations; in actual use, the data would come from an input device and could be read from the FI (Flag In) input of the ADSP-2101.

The CPFSK routine calls two external routines not shown here. The *cntlreg_inits* routine initializes the ADSP-2101's control registers. The *boot_sin* routine computes the sine of the input in AX0, returning the output in AR.

After setup (initializing variables, etc.) the processor waits for one of two interrupts. The SPORT0 interrupt causes the processor to calculate the next output sample by adding the current phase increment to the phase accumulator and computing the sine of the result. The output samples are transmitted from SPORT0 and are also sent to a DAC for display (for demonstration).

The timer interrupt causes the processor to select a new phase increment based on the value of the input data. Because the data is binary (1 or 0) it could be input through the flag input (FI) pin instead of data memory as shown. The code would have to be modified to use the state of the input flag as a condition for selecting the phase increment.



φ = current phase value (stored in "phase accumulator")

$\Delta\varphi$ = current phase increment

$\Delta\varphi_a$ = phase increment for tone a

$\Delta\varphi_b$ = phase increment for tone b

Figure 2.33 CPFSK Flow Diagram

```
.MODULE/BOOT=0/ABS=0    cpfsk_modulator;

{ CPFSK - Continuous Phase Frequency Shift Keying modulator

   input:               data stream stored in DM circ buffer (for demo)
                        in actual use, data could be state of FLAG_IN pin
   output:              dac0 - CPFSK output waveform
                        dac1 - input data stream (echoed for demo display)
                        spkr - CPFSK “sound”
}

.EXTERNAL               boot_sin;
.EXTERNAL               cntlreg_inits;
.PORT                   write_dac0;
.PORT                   write_dac1;
.PORT                   load_dac;

.CONST                  lo_tone=220;{Hertz}
.CONST                  hi_tone=880;{Hertz}
.CONST                  logic_one=H#7F00;
.CONST                  logic_zero=0;

.VAR/CIRC               demo_input_data[7];
.VAR                    hertz0, phase_incr_0;
.VAR                    hertz1, phase_incr_1;
.VAR                    phase_accumulator, phase_increment;

{————————————————————————————————}
   JUMP start; RTI; RTI; RTI;       {Reset Vector}
   RTI; RTI; RTI; RTI;              {irq2}
   RTI; RTI; RTI; RTI;              {sport0 TX}
   JUMP sample; RTI; RTI; RTI;      {sport0 RX} {at 8 kHz rate}
   RTI; RTI; RTI; RTI;              {irq0}
   RTI; RTI; RTI; RTI;              {irq1}
   CALL symbol; RTI; RTI; RTI;      {timer}     {at 2400 baud}

{————————————————————————————————}
start:  CALL cntlreg_inits;         {set up SPORTS, TIMER, etc}
        M7=1; L7=0;                 {used by bootsin routine}

{————————————————————————————————}
baud_clock:       L0=0;
                  M0=1;
                  I0=H#3FFB;           {point to DM-mapped TIMER ctrl regs}
                                       {2400 baud=5120 cycles @ 12.288 MHz}
{H#3FFB}          DM(I0,M0)=0;                 {TIMER - TSCALE}
{H#3FFC}          DM(I0,M0)=5119;              {TIMER - TPERIOD}
{H#3FFD}          DM(I0,M0)=5119;              {TIMER - TCOUNT}
```

(listing continues on next page)

```
{————————————————————————————————}
make_demo_data:   SI=lo_tone;            DM(hertz0)=SI;
                  SI=hi_tone;            DM(hertz1)=SI;
                  SI=logic_one;          DM(demo_input_data)=SI;
                  SI=logic_zero;         DM(demo_input_data+1)=SI;
                                         DM(demo_input_data+2)=SI;
                                         DM(demo_input_data+3)=SI;
                                         DM(demo_input_data+4)=SI;
                                         DM(demo_input_data+5)=SI;
                                         DM(demo_input_data+6)=SI;
                  I0=^demo_input_data;
                  L0=%demo_input_data;

{———————————————————————————————}
{These segments convert “Hertz” to 8 kHz Phase_Increment}

load_tone1:       SI=DM(hertz1);
                  SR=ASHIFT SI BY 3(HI);
                  MY0=H#4189;            {mult Hz by .512*2}
                  MR=SR1*MY0(RND);       {i.e. mult by 1.024}
                  SR=ASHIFT MR1 BY 1(HI);
                  DM(phase_incr_1)=SR1;

load_tone0:       SI=DM(hertz0);
                  SR=ASHIFT SI BY 3(HI);
                  MY0=H#4189;            {mult Hz by .512*2}
                  MR=SR1*MY0(RND);       {i.e. mult by 1.024}
                  SR=ASHIFT MR1 BY 1(HI);
                  DM(phase_incr_0)=SR1;

{————————————————————————————————}
        SI=0;
        DM(phase_accumulator)=SI;   {clear phase accumulator on startup}
        CALL symbol;                {start with first symbol}
        ICNTL=B#01111;
        IMASK=B#001001;             {enable SPORT0_RX, TIMER now}
        ENA TIMER;                  {start baud_clock now}

{————————————————————————————————}
here:   JUMP here;      {wait for symbol and sample interrupts}
```

```
{============================================================}
{========== P R O C E S S   A   N E W   S A M P L E ==========}
{============================================================}
sample: AX0=DM(phase_accumulator);
        AY0=DM(phase_increment);
        AR=AX0+AY0;
        DM(phase_accumulator)=AR;
        AX0=AR;
        CALL boot_sin;
sound:  DM(write_dac0)=AR;          {“display” CPFSK on oscilloscope}
        DM(load_dac)=AR;
        SR=ASHIFT AR BY -2(HI);
        TX0=SR1;                    {“hear” CPFSK from speaker (PCM out)}
        RTI;


{============================================================}
{========== P R O C E S S   A   N E W   S Y M B O L ==========}
{============================================================}
symbol: AX1=DM(I0,M0);              {get input data (could be FLAG_IN)}
        DM(write_dac1)=AX1;         {echo input data stream for demo}
        DM(load_dac)=AR;
        AF=PASS AX1;
        IF EQ JUMP zero;
one:    SI=DM(phase_incr_1);        DM(phase_increment)=SI; RTS;
zero:   SI=DM(phase_incr_0);        DM(phase_increment)=SI; RTS;

.ENDMOD;
```

Listing 2.21 CPFSK Program (ADSP-2101)

2.7 V.27 *ter* & V.29 MODEM TRANSMITTERS

V.27 *ter* and V.29 modem transmitters are often used in facsimile transmission systems. This section contains example programs for implementing these transmitters. The subroutines for each transmitter are listed at the end of each subsection.

These subroutines include a V.27 *ter* transmitter and a V.29 transmitter with two V.29 fallback modes. The code includes the scrambler, the IQ encoder, pulse-shaped filter and modulator.

The V.27 *ter* scrambler does not implement the extra functions required to check for repeating patterns. Both transmitters use the same pulse-shaped filter code and random number generator code. These listing are included only in section 2.7.1.

The code is contained in two subdirectories, V.27 and V.29. Each directory contains the code and ancillary files necessary for that demonstration. The file MAINXX.DSP calls a random number generator that creates data to be transmitted. The file BUILD.BAT assembles and links the files.

The demonstration code is configured for an ADSP-2101 EZ-LAB® Evaluation Board. To observe the encoder constellation, attach an analog oscilloscope in X-Y configuration to the DAC0 and DAC1 pins on the EZ-Lab board. To observe the eye pattern, attach an oscilloscope probe (in sweep mode) to the analog output pin of the codec. A synchronization pulse is available on DAC2 of the EZ-LAB board. The V.29 demonstration enters the fallback modes when the IRQ2 button is pressed.

2.7.1 V.27 *ter* Transmitter

The CCITT Recommendation is a 4800 bits/s modem standard for data transmission over the general switched telephone network. The recommendation defines the following characteristics:

- Data rate of 4800 bits/s with 8-phase differentially encoded modulation
- Fallback mode of 2400 bits/s with a 4-phase differentially encoded modulation signal
- Provision for backward channel with a modulation rate of 75 bauds

- An adaptive equalizer in the receiver
- The carrier frequency is 1800 Hz ± 1 Hz

CCITT also specifies a raised cosine filter on the transmitter and the receiver, a data scrambler, the phase encoding and the turn on sequence.

Figure 2.34 is a block diagram of a modem transmitter. The data stream to be transmitted is first scrambled to randomize the data. The data scrambler has a generating polynomial of the form:

$$1 + x^{-6} + x^{-7}$$

with additional logic to guard against repeating patterns.

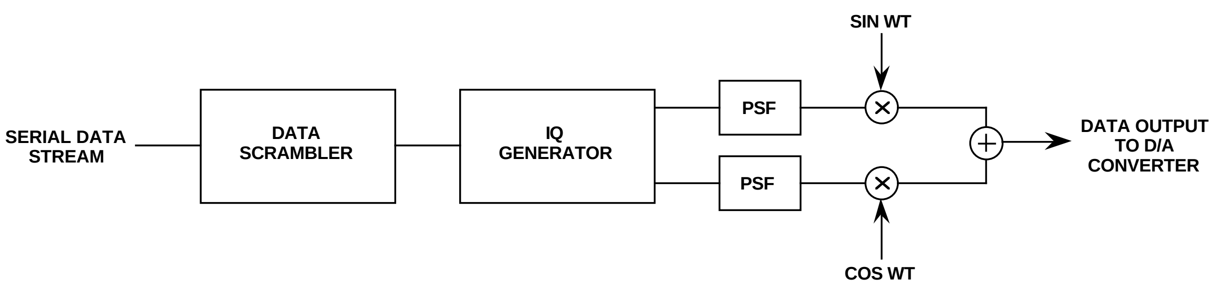


Figure 2.34 Modem Transmitter Block Diagram

The scrambled data stream is divided into groups of three bits (tribits) and is encoded as a phase change relative to the phase of the preceding word transmitted. The phase change for each tribit combination is shown in Table 2.5.

Tribit Values			*Phase Change*
0	0	1	0°
0	0	0	45°
0	1	0	90°
0	1	1	135°
1	1	1	180°
1	1	0	225°
1	0	0	270°
1	0	1	315°

Table 2.5 8-Point V.27 *ter* Phase Changes

The output of the encoder is then mapped to one point in an 8-point signal space, or constellation. The signal space mapping produces two coordinates, one for the real part and one for the imaginary part of a quadrature amplitude modulator. Figure 2.35 shows the 8-point constellation.

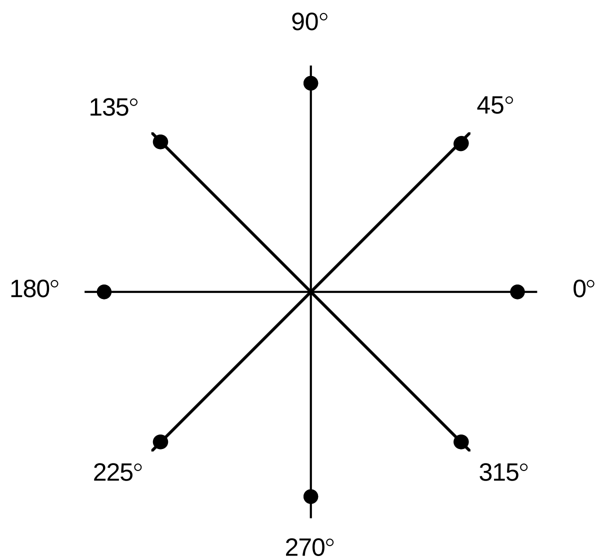


Figure 2.35 8-Point V.27 *ter* Constellation

The output of the signal mapping is interpolated to a 9.6 kHz sample rate and pulse-shaped filtered to assure zero inter-symbol interference. The signal is then modulated onto an 1800 Hz carrier and then sent to the DAC for transmission.

The 2400 bits per second fallback mode is similar to the 4800 bits per second mode. The scrambled data stream is divided into groups of two bits (dibits) before they are encoded into phase information. Table 2.6 and Figure 2.36 show the phase change table and constellation of the 4-point signal space used in the mode.

Dibit Values		*Phase Change*
0	0	0°
0	1	90°
1	1	180°
1	0	270°

Table 2.6 4-Point V.27 *ter* Phase Changes

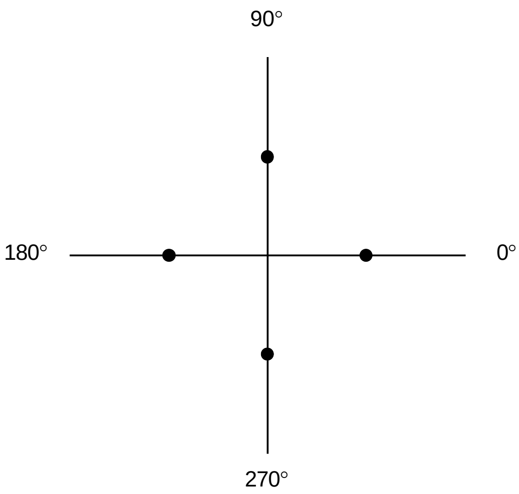


Figure 2.36 4-Point V.27 *ter* Constellation

```
{Main routine for V.27 ter Modem                                               }
{Analog Devices                                                                }
{DSP Applications                                                              }
{                                                                              }
.module/ram/boot=0/abs=0      V27_MOD;
{────────────External Function Declaration───────────────────────────────────}
.external get1;
.external scramble;
.external iq;
.external psf;
.external rand;
{────────────Global Variable Declaration─────────────────────────────────────}
                                    {For Baud/Sample Count                     }
.var/dm           sample_count;     {Number of samples until next              }
                                    {baud. Default is ? between baud           }
                                    {For Scrambler                             }
.var/dm           seed_hi;          {Seeds for random number generator         }
.var/dm           seed_lo;
.var/dm           tri_bit;          {Next bit to transmit                      }
.var/dm           bit_count;        {Number of bits left in current word       }
.var/dm/ram/circ  buffer[12];       {Delay Buffer                              }
.var/dm           buf_ptr1;         {Pointer to y(n-6) buffer value            }
.var/dm           buf_ptr2;         {Pointer to y(n-7) buffer value            }
                                    {For IQ Generator                          }
.var/pm/ram       delta_phi[8];     {Tribit phase change table                 }
.var/dm/ram       phi_value;        {Current phi value                         }
.var/pm/ram       I_table[8];       {Tables for IQ Output Values               }
.var/pm/ram       Q_table[8];
```

(listing continues on next page)

```
                                    {For Pulse Shaped FIlters                }
.var/dm/ram/circ  i_delay_line[5];  {Data Delay line for I channel PSF       }
.var/dm/ram       i_delay_ptr;      {Oldest value in delay line              }
.var/pm/ram/circ  i_coef_line[17];  {Coefficients for I challen PSF          }
.init i_coef_line : <17.dat>;
.var/dm/ram       i_coef_ptr;       {Last Coeff read by PSF                  }
.var/dm/ram       i_psf_output;     {Last output value of PSF                }
                                    {For DAC Outputs                         }
.var/dm/ram       sync;             {Sync value for DAC                      }
.var/dm/ram       i_output;         {Pulse shaped filter output value        }
.var/dm/ram       i_value;          {IQ section I output                     }
.var/dm/ram       q_value;          {IQ section Q output                     }

.init delta_phi:  0x200, 0x000, 0x400, 0x600,
                  0xC00, 0xE00, 0xA00, 0x800;
.init I_table:    0x7fff00, 0x5b0000, 0x000000, 0xa40000,
                  0x800000, 0xa40000, 0x000000, 0x5b0000;
.init Q_table:    0x000000, 0x5b0000, 0x7fff00, 0x5b0000,
                  0x000000, 0xa40000, 0x800000, 0xa40000;
.port    write_dac0;
.port    write_dac1;
.port    write_dac2;
.port    write_dac3;
.port    load_dac;
.global  delta_phi;
.global  phi_value;
.global  I_table;
.global  Q_table;
.global  buffer;
.global  buf_ptr1;
.global  buf_ptr2;
.global  tri_bit;
.global  bit_count;
.global  i_delay_line;
.global  i_delay_ptr;
.global  i_coef_line;
.global  i_coef_ptr;
.global  i_psf_output;
.global  sync;
.global  i_output;
.global  i_value;
.global  q_value;
.global  seed_hi;
.global  seed_lo;
```

```
{—————————————————Interrupt Vectors———————————————————————————————————————}
RESETV:  jump start;          {Reset Vector     }
         rti; rti; rti;
IRQ2V:   rti; rti; rti; rti;  {IRQ2 Vector                               }
HIPWV:   rti; rti; rti; rti;  {Hip Write Interrupt Vector                }
HIPRV:   rti; rti; rti; rti;  {Hip Read Interrupt Vector                 }
SPRT0T:  rti; rti; rti; rti;  {Sport0 Transmit Interrupt Vector          }
SPRT0R:  jump next_output;    {Sport0 Receive Interrupt Vector           }
         rti; rti; rti;
IRQ1V:   rti; rti; rti; rti;  {IRQ1 Interrupt Vector                     }
IRQ0V:   rti; rti; rti; rti;  {IRQ0 Interrupt Vector                     }
TIMERV:  rti; rti; rti; rti;  {Timer Interrupt Vector                    }
{————————————————Main Code Starts Here————————————————————————————————————}
start:   call V27_INIT;       {Initialize ADSP-2101 Dags and Sports      }
         imask = 0x08;        {Enable Sports and Interrupts              }
         ax0 = 0x101f;
         dm(0x3fff) = ax0;
fevr:    idle;                {Wait for Interrupt                        }
         jump fevr;
{—————————————————MAIN ROUTINE————————————————————————————————————————————}
next_output:
         {This interrupt routine is executed every output sample rate  }
         {One output sample is calculated and a new baud is calulated  }
         {if interrupt occurs at baud rate                             }
                              {Output values to the DAC                  }
         ax0 = dm(i_value);
         dm(write_dac0) = ax0;
         ax0 = dm(q_value);
         dm(write_dac1) = ax0;
         ax0 = dm(sync);
         dm(write_dac2) = ax0;
         si = dm(i_output);
         sr = ashift si by -2 (lo);
         dm(write_dac3) = sr0;
         dm(load_dac) = sr0;
         tx0 = sr0;

         {Check if new baud should be calculated                       }
         ax0 = dm(sample_count);
         ay0 = 1;
         my1 = 0x7fff;        {Set sync value for new sample             }
         dm(sync) = my1;
         ar = ax0 - ay0;
         dm(sample_count) = ar;
         if gt jump next_sample;
```

(listing continues on next page)

```
next_baud:
        i4 = ^i_coef_line;   {Reset Coefficient Pointer               }
        dm(i_coef_ptr) = i4;
        ax0 = 4;             {Reset sample counter                    }
        dm(sample_count) = ax0;
        si = 0;              {Zero SI Register }
        cntr = 3;
        do scram_bit until ce;
           call get1;        {Get next 3 data bits to be transmitted  }
           call scramble;
scram_bit:
        my1 = 0x8000;        {Set sync value for new baud             }
        dm(sync) = my1;
        call IQ;             {Calculate delta phi,phase sum,I and Q   }
        jump next_sample;

        {Generate Next output sample to DAC    }
next_sample:
        call psf;            {Low Pass Filter I and Q components      }
done:   rti;

{———————————Initialization for V.27————————————————————————————————————}
V27_INIT:
                        {———————SPORT INIT——————————————————————————————}
ax0 = 0x02;             {SCLKDIV = 2.048 MHz                            }
        dm(0x3ff5) = ax0;
        ax0 = 213;      {RFSDIV = 213 for 9600                          }
        dm(0x3ff4) = ax0;
        ax0 = 0x6b27;   {Internal SCLK, Frame Syncs                     }
        dm(0x3ff6) = ax0;
                        {———————Wait States—————————————————————————————}
        ax0 = 0xffff;
        dm(0x3ffe) = ax0;

                        {————————Data Buffer Inits——————————————————————}
        I0 = ^buffer;
        m0 = 1;
        L0 = %buffer;
        cntr = %buffer;
        ax0 = 0;
        do zloop until ce;
 zloop:    dm(i0,m0) = ax0;

                        {—————Variable Initializations——————————————————}
        ax0 = 0;
        dm(tri_bit) = ax0;
        dm(bit_count) = ax0;
        dm(phi_value) = ax0;
        dm(sample_count) = ax0;
```

```
        ax0 = ^buffer+5;
        dm(buf_ptr1) = ax0;
        ax0 = ^buffer+6;
        dm(buf_ptr2) = ax0;
        ax0 = 0xa5a5;
        dm(seed_hi) = ax0;
        ax0 = 0x1234;
        dm(seed_lo) = ax0;

                        {———————Pulse Shaped Filter Inits———————————————}
        i0 = ^i_coef_line;
        dm(i_coef_ptr) = i0;
        i0 = ^i_delay_line;
        dm(i_delay_ptr) = i0;
        l0 = %i_delay_line;
        m0 = 1;
        ax0 = 0;
        cntr = %i_delay_line;
        do dloop until ce;
dloop:
        dm(i0,m0) = ax0;
        ax0 = 0x7fff;
        dm(sync) = ax0;

                        {—————————DAG INIT——————————————————————————————}
        m0 = 0;         {Init M0,M1,M5,M5                               }
        m1 = 1;
        m4 = 0;
        m5 = 1;
        l0 = 0;         {Set All L registers to zero                    }
        l1 = 0;
        l2 = 0;
        l3 = 0;
        l4 = 0;
        l5 = 0;
        l6 = 0;
        l7 = 0;
        rts;
{———————————————END INITIALIZATION————————————————————————————————————————}

.endmod;
```

Listing 2.22 Main V.27 *ter* Routine (MAIN27.DSP)

```
{GET2 routine for v.27 Modem                                                  }
{Analog Devices                                                               }
{DSP Applications                                                             }
{                                                                             }
.module/ram/boot=0      get1mod;
{Get 1 data bit to be transmitted                                             }
{                                                                             }
{  INPUTS:                                                                    }
{        dm(data_word)  16 bit words for transmission                         }
{        dm(bit_count)  number of bits left in word                           }
{        m0 = 0                                                               }
{        m1 = 1                                                               }
{        m4 = 0                                                               }
{        m5 = 1                                                               }
{                                                                             }
{  OUTPUTS:                                                                   }
{        dm(tri_bit):   bit to transmit                                       }
{                                                                             }
{  USAGE:                                                                     }
{        AX0, AY0, AR, SI, SR                                                 }
{                                                                             }
{Global Variable Declaration                                                  }
.external tri_bit;   {bit to transmit                                         }
.external bit_count; {number of bits left in word                             }

{Local Variable Declaration                                                   }
.var/dm/ram        data_word;          {Data word to be transmitted           }
.external          rand;
{————————————————Code Start———————————————————————————————————————————————————}
.entry   GET1;
GET1:
         ay0 = dm(bit_count);          {If bit_count is zero                  }
         ar = pass ay0;                {load new word                         }
         if ne jump dec_count;
         call rand;                    {Load new data word here               }
         dm(data_word) = ay0;
         ay0 = 16;
dec_count:
         ar = ay0 -1;                  {Decrement counter                     }
         dm(bit_count) = ar;
get_bit:
         sr0 = dm(data_word);          {Get next bit                          }
         ay0 = 0x01;
         ar = sr0 and ay0;
         dm(tri_bit) = ar;
         sr = lshift sr0 by -1 (lo);
         dm(data_word) = sr0;          {Write data_word for next time         }
         rts;
.endmod;
```

Listing 2.23 Data Acquisition Routine (GET27.DSP)

```
{DSP Applications                                                              }
{                                                                              }
{ 2/20/91       Start Date                                                     }
{This module performs v.27 ter scrambling on one input bit                     }
{                                                                              }
{The scrambler generating polynomial is xin + y(n-6) + y(n-7)                  }
{V.27 ter specifies additional checking for certain pattarns.                  }
{This has not been implemented.                                                }
{                                                                              }
{  INPUTS:                                                                     }
{        dm(tri_bit) = bit to be scrambled                                     }
{        si = partial tri bit                                                  }
{                                                                              }
{  OUTPUTS:                                                                    }
{        si = scrambled output bit                                             }
{             si is left shifted by 1 and the new bit is put in b0             }
{                                                                              }
{  USAGE:                                                                      }
{        I0 = ^buffer                                                          }
{        I1 = Y(n-6)                                                           }
{        I2 = y(n-7)                                                           }
{                                                                              }
.module/ram/boot=0       SCRAMBLE_MOD;
{Local variable Declarations                                                   }
.external   buffer;                     {Delay Buffer                          }
.external   buf_ptr1;                   {Pointer to y(n-6) buffer value        }
.external   buf_ptr2;                   {Pointer to y(n-7) buffer value        }
.external   tri_bit;
.entry      SCRAMBLE;
SCRAMBLE:
   i1 = dm(buf_ptr1);             {Get Pointer values                          }
   i2 = dm(buf_ptr2);
   m3 = -1;
   L1 = %buffer;                  {Set circular buffer registers               }
   L2 = %buffer;

   ax0 = dm(tri_bit);             {Get next bit to scramble                    }
   ay0 = dm(i1,m3);               {y(n-6)                                      }
   ar = ax0 xor ay0, ay0=dm(i2,m0); {y(n-7)                                    }
   ar = ar xor ay0;               { = y(n)                                     }
   dm(i2,m3) = ax0;               {Store Oldest Value                          }

   sr = lshift si by 1 (lo);      {Store output in sr0                         }
   sr = sr or lshift ar by 0 (lo);
   si = sr0;

   dm(buf_ptr1) = i1;             {Save buffer pointers                        }
   dm(buf_ptr2) = i2;
   L1 = 0;                        {Clear L registers                           }
   L2 = 0;
   rts;
.endmod;
```

Listing 2.24 Data Scrambler Routine (SCRAM27.DSP)

```
{Analog Devices                                                         }
{DSP Applications                                                       }
{ 2/20/91      Start Date                                               }
.module/ram/boot=0      PHI_MOD;
{Calculate delta phi, phase sum, I and Q{Analog Devices                 }
{                                                                       }
{  INPUTS:                                                              }
{           SI = TriBit Value                                           }
{           dm[PHI] = current Phi Value                                 }
{                                                                       }
{  OUTPUTS:                                                             }
{           MX0 = I Value                                               }
{           MX1 = Q Value                                               }
{                                                                       }
{  USAGE:                                                               }
{           I3, M3, AR, AX0, AY0                                        }
{                                                                       }
.external   Q_table;
.external   I_table;
.external   delta_phi;
.external   phi_value;
.external   i_coef_line;
.external   i_coef_ptr;
.external   i_delay_line;
.external   i_delay_ptr;
.external   i_value;
.external   q_value;
.entry      IQ;
IQ:
{Look up Delta_phi value and add to phi for new phase                   }
            I7 = ^delta_phi;
            m7 = si;
            modify(i7, m7);
            ax0 = dm(phi_value);
            ay0 = pm(i7,m4);
            ar = ax0 + ay0;
            ay1 = 0x0f;
            ar = ar and ay1;
            dm(phi_value) = ar;
{Look up I and Q components                                             }
{To find I/Q values, use phi/2 as offset into table                     }
            sr = lshift ar by -1 (lo);
            I6 = ^I_table;
            I7 = ^Q_table;
            m7 = sr0;
            modify(I6,M7);
            modify(I7,M7);
            mx0 = pm(i6,m4);
            mx1 = pm(i7,m4);
```

```
   {Write new values for DAC                                             }
         dm(i_value) = mx0;
         dm(q_value) = mx1;
   {Write new I value to PSF Delay Line                                  }
         I6 = dm(i_delay_ptr);
         L6 = %i_delay_line;
         M6 = -1;
         dm(i6,m6) = mx0;
         dm(i_delay_ptr) = i6;
         L6 = 0;
         rts;
.endmod;
```

Listing 2.25 IQ Generator Routine (IQ27.DSP)

```
{Low Pass Filter I component only                                          }
{Analog Devices                                                            }
{DSP Applications                                                          }
{                                                                          }
{        INPUTS:                                                           }
{           MX0 = I Value                                                  }
{                                                                          }
{        OUTPUTS:                                                          }
{           dm(I_LPF) = I filter output                                    }
{                                                                          }
{        USAGE:                                                            }
{           MX0, MX1, MY0, MR, I3, I7, L3, L7                              }
{                                                                          }
.module/boot=0/ram      PSF_MOD;
.external         i_delay_line;
.external         i_delay_ptr;
.external         i_coef_line;
.external         i_coef_ptr;
.external         i_output;
.entry            PSF;
PSF:
                                    {Initialize DAGs                       }
        L0 = %i_delay_line;
        L4 = %i_coef_line;
        I0 = dm(i_delay_ptr);
        I4 = dm(i_coef_ptr);
        modify(i0,m1);              {Point to newest data value            }
        m6 = 4;                     {Interpolate by factor of 4            }
                                    {I filter                              }
        mr = 0, mx0 = dm(i0,m1);    {load first data word                  }
        my0 = pm(i4,m6);            {load first coefficient                }
        cntr = 3;
        do i_loop until ce;
   i_loop:
        mr = mr+mx0*my0(SS), mx0 = dm(i0,m1), my0 = pm(i4,m6);
        mr = mr+mx0*my0(RND);
        if mv sat mr;
        dm(I_output) = mr1;
        I4 = dm(i_coef_ptr);        {Point to next set of coefficients }
        modify(i4,m5);
        dm(i_coef_ptr) = I4;
        L0 = 0;
        L4 = 0;
        rts;
.endmod;
```

Listing 2.26 Pulse Shape Filter Routine (PSF.DSP)

```
.MODULE/ram/boot=0      rand_sub;
{
            Linear Congruence Uniform Random Number Generator

   INPUTS:
            dm(Seed_hi) = MSW of seed value
            dm(Seed_lo) = LSW of seed value

   OUTPUTS:
            ay0 = random number;
            dm(Seed_hi) = MSW of updated seed value
            dm(Seed_lo) = LSW of updated seed value
}

.external   seed_hi;
.external   seed_lo;

.ENTRY      rand;
rand:       MY1=25;                             {Upper half of a}
            MY0=26125;                          {Lower half of a}
            sr1 = dm(seed_hi);
            sr0 = dm(seed_lo);
            mr=sr0*my1(uu);                     {A(HI)*X(LO)}
            mr=mr+sr1*my0(uu);                  {A(HI)*X(LO) + A(LO)*X(HI)}
            mr2=mr1;
            mr1=mr0;
            {mr2=si;}
            mr0=h#fffe;                         {C=32767, LEFT-SHIFTED BY 1}
            mr=mr+sr0*my0(uu);                  {(ABOVE) + A(LO)*X(LO) + C}
            sr=ashift mr2 by 15 (hi);
            sr=sr or lshift mr1 by -1 (hi);     {RIGHT-SHIFT BY 1}
            sr=sr or lshift mr0 by -1 (lo);
            dm(seed_hi) = sr1;
            dm(seed_lo) = sr0;
            ay0 = sr1;
         rts;
.endmod;
```

Listing 2.27 Random Number Generator Routine (RAND.DSP)

```
.module/ram/boot=0            MODULATE_MOD;
{Modulate and sum signals                                        }
{                                                                }
{  INPUTS:                                                       }
{        dm(i_lpf) = I LPF output                                }
{        dm(q_lpf) = Q LPF output                                }
{                                                                }
{  OUTPUTS:                                                      }
{        dm(result) = output value                               }
{                                                                }
{  USAGE:                                                        }
{        I4, M6, M7, MX0, MY0, MR, SR                            }
{                                                                }
.var/pm/ram/circ  cosine[16];       {Cosine Table                }
.var/dm/ram       cos_ptr;          {Current Pointer to Cosine Table }
.var/dm/ram       output;           {Output Value                }
.external         i_lpf;
.external         q_lpf;
.init             cosine: <cosval.dat>;
.init             cos_ptr: ^cosine;
.entry            modulator;
MODULATOR:
   cntr = 6;
   do mod_loop until ce;
         i4 = dm(cos_ptr);                {Initialize DAGs            }
         m6 = -4;
         m7 = 7;
         L4 = %cosine;
         mx0 = pm(i4,m6);                 {Read Cosine, point to sine  }
         my0 = dm(i_lpf);                 {Read I value               }
         mr = mx0*my0(ss), mx0=pm(i4,m7); {cos(k)*I(k), get -sin      }
         my0 = dm(q_lpf);                 {Read Q Value               }
         mr = mr + mx0*my0(rnd);          {-Q * Sine                  }
         dm(cos_ptr) = i4;                {Save cosine Pntr           }
         sr = ashift mr2 by -1 (HI);      { Scale output by 1/2       }
         sr = sr or lshift mr1 by -1 (lo);
mod_loop:
         dm(output) = sr0;
         L4 = 0;
   rts;
.endmod;
```

Listing 2.28 Signal Modulation Routine (MODULATE.DSP)

2.7.2 V.29 Transmitter

The CCITT Recommendation V.29 is a 9600 bits per second modem standard for data transmission over the general switched telephone network. The specification defines the following characteristics:

- Data rate of 9600 bits/s with 8-phase differentially encoded modulation
- Fallback rates of 7200 and 4800 bits/s
- Provision for backward channel with a modulation rate of 75 bauds
- An adaptive equalizer in the receiver
- The carrier frequency is 1700 Hz ± 1 Hz

CCITT also specifies a raised cosine filter on the transmitter and the receiver, a data scrambler, the phase encoding and the turn on sequence.

The data stream to be transmitted is first scrambled to randomize the data. The data scrambler has a generating polynomial of the form:

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

The scrambled data is divided into for bits (quadbits). The first bit (Q1) is used to determine the signal amplitude and the remaining three bits (Q2, Q3, and Q4) are encoded as a phase change relative to the phase of the preceding word transmitted. Table 2.7 and Figure 2.37 show the phase change table and the V.29 8-point constellation.

Q2	*Q3*	*Q4*	*Phase Change*
0	0	1	0°
0	0	0	45°
0	1	0	90°
0	1	1	135°
1	1	1	180°
1	1	0	225°
1	0	0	270°
1	0	1	315°

Table 2.7 8-Point V.29 Phase Changes

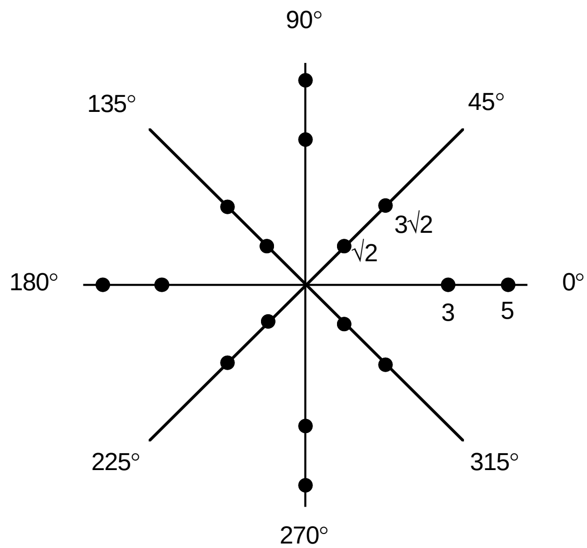


Figure 2.37 V.29 Constellation

The output of the signal mapping is interpolated to a 9.6 kHz sample rate and pulse-shaped filtered to assure zero inter-symbol interference. The signal is then modulated onto a 1700 Hz carrier and then sent to the DAC for transmission.

The 7200 bits per second fallback mode is similar to the 9600 bits per second mode. The scrambled data is divided into groups of three bits (tribits). Table 2.7 can be used to determine the phase change. The first

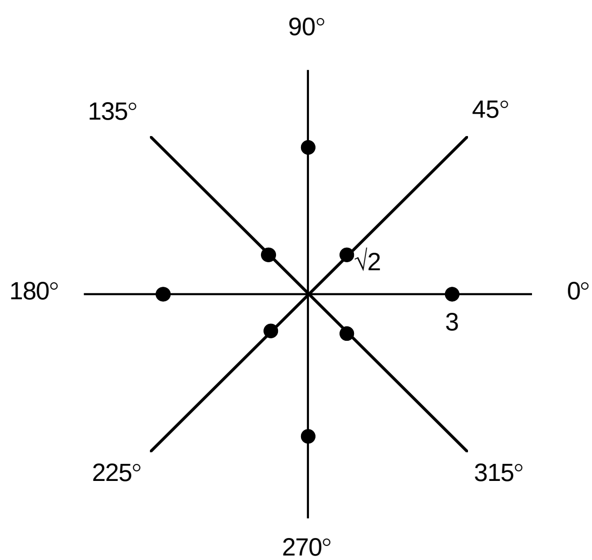


Figure 2.38 V.29 Constellation For 7200 bits/s Fallback Mode

tribit represents Q2 in the Table 2.7 and the next two tribits represent Q3 and Q4 respectively. Q1 is assumed to be zero. Figure 2.38 shows the 8-point V.29 constellation for the 7200 bits/s fallback mode.

The 4800 bits per second fallback mode is similar to the 4800 bits per second mode. The scrambled data stream is divided into groups of two bits (dibits). Table 2.7 can be used to determine the phase change. The tribits represent bit Q2 and Q3 in the figure. Bit Q4 in Table 2.7 is the inversion of the mod 2 sum of the two data bits and Q1 is assumed to be zero. Figure 2.39 shows the constellation for this mode.

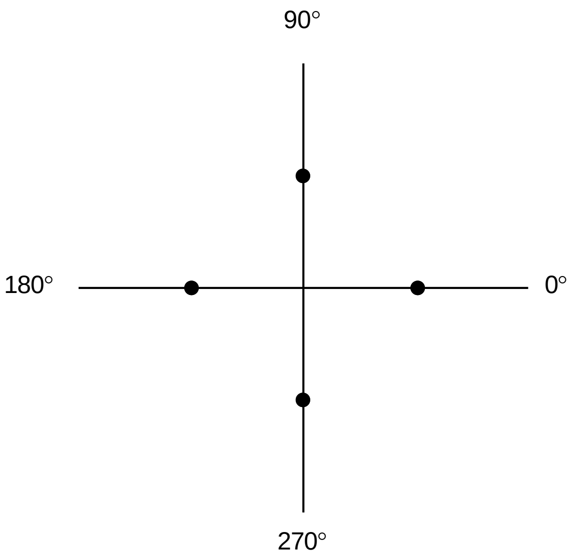


Figure 2.39 V.29 Constellation For 4800 bits/s Fallback Mode

```
{Main routine for V.29 ter Modem                                        }
{Analog Devices                                                         }
{DSP Applications                                                       }
{                                                                       }
{ 2/20/91   Start Date                                                  }
.module/boot=0/ram/abs=0      29_MOD;
{────────────External Function Declaration─────────────────────────────}
.external get1;
.external scramble;
.external IQ;
.external psf;
.external rand;
{────────────Global Variable Declaration───────────────────────────────}
                              {For baud/sample count                    }
.var/dm/ram       sample_count;  {Number of samples until next baud    }
                              {Default is 3 samples between each baud  }
                              {For Scrambler                            }
.var/dm/ram       seed_hi;    {Seeds for random number generator       }
.var/dm/ram       seed_lo;
.var/dm/ram       quad_bit;   {next bit to transmit                     }
.var/dm/ram       bit_count;  {Number of bits left in current word      }
.var/dm/ram/circ  buffer[23]; {Scrambler Delay Buffer                   }
.var/dm           buf_ptr2;   {Pointer to y(n-18) buffer value          }
.var/dm           buf_ptr3;   {Pointer to y(n-23) buffer value          }
                              {For IQ Generator                         }
.var/pm/ram       delta_phi[8];  {Quadbit phase change table            }
.var/dm/ram       phi_value;     {Current phi value                     }
.var/pm/ram       I_table_0[8];  {Tables for IQ Output Values           }
.var/pm/ram       Q_table_0[8];
.var/pm/ram       I_table_1[8];
.var/pm/ram       Q_table_1[8];
.var/dm/ram       fallback_mode;    {0=9600bps, 3=7200bps, 6=4800bps   }
.var/dm/ram       fallback_count;   {Number of bits to scramble         }
                                    {For Pulse Shaped FIlters           }
.var/dm/ram/circ  i_delay_line[5];  {Data Delay line for I channel PSF }
.var/dm/ram       i_delay_ptr;      {Oldest value in delay line         }
.var/pm/ram/circ  i_coef_line[17];  {Coefficients for I challen PSF    }
.init             i_coef_line: <17.dat>;
.var/dm/ram       i_coef_ptr;       {Last Coeff read by PSF             }
.var/dm/ram       i_psf_output;     {Last output value of PSF           }
                                    {For DAC Outputs                    }
.var/dm/ram       sync;             {Sync value for DAC                 }
.var/dm/ram       i_output;         {Pulse shaped filter output value  }
.var/dm/ram       i_value;          {IQ section I output                }
.var/dm/ram       q_value;          {IQ section Q output                }
```

```
.init delta_phi:  0x000200, 0x000000, 0x000400, 0x000600,
                  0x000C00, 0x000E00, 0x000A00, 0x000800;

.init I_table_1:  0x7fff00, 0x5bff00, 0x000000, 0xa40000,
                  0x800000, 0xa40000, 0x000000, 0x5bff00;
.init Q_table_1:  0x000000, 0x5bff00, 0x7fff00, 0x5bff00,
                  0x000000, 0xa40000, 0x800000, 0xa40000;

.init I_table_0:  0x4c0000, 0x1e0000, 0x000000, 0xe10000,
                  0xb30000, 0xe10000, 0x000000, 0x1e0000;
.init Q_table_0:  0x000000, 0x1e0000, 0x4c0000, 0x1e0000,
                  0x000000, 0xe10000, 0xb30000, 0xe10000;

{————————————Ports and Global ——————————————————————————————————}
.port    write_dac0;
.port    write_dac1;
.port    write_dac2;
.port    write_dac3;
.port    load_dac;
.global  delta_phi;
.global  phi_value;
.global  I_table_1;
.global  Q_table_1;
.global  I_table_0;
.global  Q_table_0;
.global  buffer;
.global  buf_ptr2;
.global  buf_ptr3;
.global  quad_bit;
.global  bit_count;
.global  fallback_mode;
.global  i_delay_line;
.global  i_delay_ptr;
.global  i_coef_line;
.global  i_coef_ptr;
.global  i_psf_output;
.global  sync;
.global  i_output;
.global  i_value;
.global  q_value;
.global  seed_hi;
.global  seed_lo;
```

(listing continues on next page)

```
{————————————————Interrupt Vectors———————————————————————————————————————}
RESETV:  jump start;              {Reset Vector                          }
         rti; rti; rti;
IRQ2V:   jump set_fall_back;      {IRQ2 Vector                           }
         rti; rti; rti;
HIPWV:   rti; rti; rti; rti;      {Hip Write Interrupt Vector            }
HIPRV:   rti; rti; rti; rti;      {Hip Read Interrupt Vector             }
SPRT0T:  rti; rti; rti; rti;      {Sport0 Transmit Interrupt Vector      }
SPRT0R:  jump next_output;        {Generate Next Sample                  }
         rti; rti; rti;
IRQ1V:   rti; rti; rti; rti;      {IRQ1 Interrupt Vector                 }
IRQ0V:   rti; rti; rti; rti;      {IRQ0 Interrupt Vector                 }
TIMERV:  jump set_fall_back;      {Timer Interrupt Vector                }
         rti; rti; rti;
{———————————————Main Code Starts Here————————————————————————————————————}
start:   call V29_INIT;       {Initialize ADSP-2101 Dags and Sports      }
         ena timer;
         ax0 = 0x101f;        {Enable SPORT0                             }
         dm(0x3fff) = ax0;
         imask = 0x88;        {Enable Timer,Sport0,IRQ2 Interrupts       }
fevr:    idle;                {Wait for Interrupt                        }
         jump fevr;
{————————————————MAIN ROUTINE————————————————————————————————————————————}
next_output:
   {This interrupt routine is executed every 8 kHz.                     }
   {One sample is output to codec and a new Baud is calculate if        }
   {interrupt occurs at baud rate                                       }

                              {Output values to the DAC                 }
         ax0 = dm(i_value);
         dm(write_dac0) = ax0;
         ax0 = dm(q_value);
         dm(write_dac1) = ax0;
         ax0 = dm(sync);
         dm(write_dac2) = ax0;
         si = dm(i_output);
         sr = ashift si by -2 (lo);
         dm(write_dac3) = sr0;
         dm(load_dac) = sr0;
         tx0 = sr0;

         {Check if new baud should be calculated                        }
         ax0 = dm(sample_count);
         ay0 = 1;
         my1 = 0x7fff;        {Set sync value for new sample             }
         dm(sync) = my1;
         ar = ax0 - ay0;
         dm(sample_count) = ar;
         if gt jump next_sample;
```

```
next_baud:
        i4 = ^i_coef_line;   {Reset Coefficient Pointer               }
        dm(i_coef_ptr) = i4;
        ax0 = 4;             {Reset sample counter                    }
        dm(sample_count) = ax0;
        si = 0;              {Zero SI Register                        }
        cntr = dm(fallback_count);
        do scram_bit until ce;
           call get1;        {Get next N data bits to be transmitted  }
           call scramble;
scram_bit:
        ax0 = dm(fallback_mode);  {Test if in 4800 bps mode           }
        ay0 = 0x06;
        ar = ax0 - ay0;
        if ne jump not48;

mode48:                            {Calculate Q4 = inv(Q3 + Q2)       }
        sr = lshift si by -1(lo);  {sr0 = Q2                          }
        ay0 = si;
        ar = sr0 xor ay0;
        ar = not ar;
        ay1 = 1;
        ar = ar and ay1;           {AY1 must be preset to 1           }
        sr = lshift si by 1 (lo);  {Store output in si                }
        sr = sr or lshift ar by 0 (lo);
        si = sr0;
not48:
        call IQ;             {Calculate delta phi,phase sum,I and Q   }
        my1 = 0x8000;        {Set sync value for new baud             }
        dm(sync) = my1;
        jump next_sample;
next_sample:
        call psf;            {Low Pass Filter I and Q components      }
                             {Output values to the DAC                }
{       call modulator;      Modulate and sum signals                 }
done:   rti;

{————————————Change Fallback Mode———————————————————————————————————————}
set_fall_back:
        ena sec_reg;
        ax0 = dm(fallback_mode);
        mx0 = i4;                  {store i4 in temporary location    }
        mx1 = m4;                  {store m7 in temporary location    }
        my0 = L4;                  {store L4 in temporary location    }
        l4 = 0;
        m4 = ^jump_table;          {CASE Statement                    }
        i4 = ax0;
        modify(i4,m4);
        jump (i4);
```

(listing continues on next page)

```
jump_table:
   {Current fallback = 9600 bps, change to 7200bps                   }
         ay0 = 0x03;
         ax0 = 0x03;
         jump case_end;

   {Current fallback = 7200 bps, change to 4800bps                   }
         ay0 = 0x06;
         ax0 = 0x02;
         jump case_end;

   {Current fallback = 4800 bps, change to 7200bps                   }
         i4 = ^buffer;       {Reset Delay Buffer and Data Pointers  }
         l4 = %buffer;
         m4 = 1;
         ax0 = 0;
         dm(bit_count) = ax0;
         cntr = %buffer;
         do z2 until ce;
z2:      dm(i4,m4) = ax0;
         ay0 = 0x0;
         ax0 = 0x04;
         jump case_end;
case_end:
         dm(fallback_count) = ax0;
         dm(fallback_mode) = ay0;
         ax0 = 0;                        {Zero Phi Value             }
         dm(phi_value) = ax0;
         i4 = mx0;                       {Restore i4                 }
         m4 = mx1;                       {Restore m7                 }
         L4 = my0;                       {Restore L4                 }
         rti;
{———————————Initialization for V.29———————————————————————————————————}
V29_INIT:
                        {———————SPORT INIT———————————————————————————}
         ax0 = 0x02;         {SCLKDIV = 2.048 MHz                   }
         dm(0x3ff5) = ax0;
         ax0 = 213;          {RFSDIV = 639 for 9600                 }
         dm(0x3ff4) = ax0;
         ax0 = 0x6b27;       {Internal SCLK, Frame Syncs            }
         dm(0x3ff6) = ax0;
                        {———————Wait States——————————————————————————}
         ax0 = 0xffff;
         dm(0x3ffe) = ax0;
```

```
                              {————————Data Buffer Inits———————————————————}
         i0 = ^buffer;
         l0 = %buffer;
         m1 = 1;
         ax0 = 0;
         cntr = %buffer;
         do zl until ce;
zl:      dm(i0,m1) = ax0;
                              {—————Variable Initializations———————————————}
ax0 = 0;
         dm(quad_bit) = ax0;
         dm(bit_count) = ax0;
         dm(phi_value) = ax0;
         dm(sample_count) = ax0;
         ax0 = 6;
         dm(fallback_mode) = ax0;
         ax0 = 2;
         dm(fallback_count) = ax0;
         ax0 = ^buffer+17;
         dm(buf_ptr2) = ax0;
         ax0 = ^buffer+22;
         dm(buf_ptr3) = ax0;
         ax0 = 0xa5a5;
         dm(seed_hi) = ax0;
         ax0 = 0x1234;
         dm(seed_lo) = ax0;

                              {——————Pulse Shaped Filter Inits—————————————}
         i0 = ^i_coef_line;
         dm(i_coef_ptr) = i0;
         i0 = ^i_delay_line;
         dm(i_delay_ptr) = i0;
         l0 = %i_delay_line;
         m0 = 1;
         ax0 = 0;
         cntr = %i_delay_line;
         do dloop until ce;
   dloop:
         dm(i0,m0) = ax0;
         ax0 = 0x7fff;
         dm(sync) = ax0;
                              {————————Timer Init——————————————————————————}
ax0 = 0xffff;
         dm(0x3ffd) = ax0;
         dm(0x3ffc) = ax0;
         dm(0x3ffb) = ax0;
```

(listing continues on next page)

```
                        {————————DAG INIT————————————————————————————}
         m0 = 0;        {Init M0,M1,M5,M5                            }
         m1 = 1;
         m4 = 0;
         m5 = 1;
         l0 = 0;        {Set All L registers to zero                 }
         l1 = 0;
         l2 = 0;
         l3 = 0;
         l4 = 0;
         l5 = 0;
         l6 = 0;
         l7 = 0;
         rts;
{———————————————END INITIALIZATION———————————————————————————————————}
.endmod;
```

Listing 2.29 Main V.29 Routine (MAIN29.DSP)

```
{GET2 routine for v.29  Modem                                              }
{Analog Devices                                                            }
{DSP Applications                                                          }
{                                                                          }
.module/boot=0/ram          get1mod;
{Get 1 data bit to be transmitted                                          }
{                                                                          }
{     INPUTS:                                                              }
{        dm(data_word)  16 bit words for transmission                      }
{        dm(bit_count)  number of bits left in word                        }
{        m0 = 0                                                            }
{        m1 = 1                                                            }
{        m4 = 0                                                            }
{        m5 = 1                                                            }
{                                                                          }
{  OUTPUTS:                                                                }
{        dm(quad_bit):  bit to transmit                                    }
{                                                                          }
{     USAGE:                                                               }
{        AX0, AY0, AR, SI, SR                                              }
{                                                                          }

{Global Variable Declaration                                               }
.external   quad_bit;                         {bit to transmit             }
.external   bit_count;                        {number of bits left in word }
.port       in_port;

{Local Variable Declaration                                                }
.var/dm/ram       data_word;                  {Data word to be transmitted }
.external         rand;
{—————————————————Code Start———————————————————————————————————————————————}
.entry GET1;
GET1:
         ay0 = dm(bit_count);       {If bit_count is zero                  }
         ar = pass ay0;             {load new word                         }
         if ne jump dec_count;
         ay0 = 0xc123;              {Load new data word here               }
         call rand;
         dm(data_word) = ay0;
         ay0 = 16;
         dec_count: ar = ay0 -1;    {Decrement counter                     }
         dm(bit_count) = ar;
get_bit:
         sr0 = dm(data_word);       {Get next bit                          }
         ay0 = 0x01;
         ar = sr0 and ay0;
         dm(quad_bit) = ar;
         sr = lshift sr0 by -1 (lo);
         dm(data_word) = sr0;       {Write data_word for next time         }
         rts;
.endmod;
```

Listing 2.30 Data Acquisition Routine (GET29.DSP)

```
.module/boot=0/ram      SCRAMBLE_MOD;
{Analog Devices                                                    }
{DSP Applications                                                  }
{                                                                  }
{ 2/20/91   Start Date                                             }
{This module performs v.29 scrambling on one input bit             }
{                                                                  }
{The scrambler generating polynomial is xin + y(n-18) + y(n-23)    }
{v.29 specifies additional data pattarns on startup.               }
{This has not been implemented.                                    }
{                                                                  }
{  INPUTS:                                                         }
{        dm(quad_bit) = bit to be scrambled                        }
{        si = partial quad bit                                     }
{                                                                  }
{  OUTPUTS:                                                        }
{        si = scrambled output bit                                 }
{             si  is left shifted by 1 and the new bit is put in b0 }
{                                                                  }
{  USAGE:                                                          }
{        I0 = ^buffer                                              }
{        I1 = Y(n-18)                                              }
{        I2 = y(n-23)                                              }
{Local variable Declarations                                       }
.external   buffer;             {Delay Buffer                      }
.external   buf_ptr2;           {Pointer to y(n-18) buffer value   }
.external   buf_ptr3;           {Pointer to y(n-23) buffer value   }
.external   quad_bit;
.entry      SCRAMBLE;
SCRAMBLE:
        i1 = dm(buf_ptr2);      {Get Pointer values                }
        i2 = dm(buf_ptr3);
        m3 = -1;
        L1 = %buffer;           {Set circular buffer registers     }
        L2 = %buffer;

        ax0 = dm(quad_bit);           {Get next bit to scramble    }
        ay0 = dm(i1,m3);              {y(n-18)                     }
        ar = ax0 xor ay0, ay0=dm(i2,m0); {y(n-23)                  }
        ar = ar xor ay0;              { = y(n)                     }
        dm(i2,m3) = ax0;              {Store Oldest Value          }

        sr = lshift si by 1 (lo);     {Store output in sr0         }
        sr = sr or lshift ar by 0 (lo);
        si = sr0;

        dm(buf_ptr2) = i1;            {Save buffer pointers        }
        dm(buf_ptr3) = i2;
        L1 = 0;                       {Clear L registers           }
        L2 = 0;
        rts;
.endmod;
```


Listing 2.31 Data Scrambler Routine (SCRAM29.DSP)

```
{Analog Devices                                                   }
{DSP Applications                                                 }
{                                                                 }
{ 2/20/91       Start Date                                        }
.module/boot=0/ram      PHI_MOD;
{Calculate delta phi, phase sum, I and Q                          }
{                                                                 }
{        INPUTS:                                                  }
{           SI = QuadBit Value in four LSBs                       }
{           Q1 (Amplitude Bit) is in b3                           }
{           Q2-4 (Phase Change) is b b 2-0                        }
{           dm[PHI] = current Phi Value                           }
{                                                                 }
{        OUTPUTS:                                                 }
{           MX0 = I Value                                         }
{           MX1 = Q Value                                         }
{                                                                 }
{        USAGE:                                                   }
{           I3, M3, AR, AX0, AY0                                  }
{                                                                 }
.external   Q_table_0;
.external   I_table_0;
.external   Q_table_1;
.external   I_table_1;
.external   delta_phi;
.external   phi_value;
.external   i_coef_line;
.external   i_coef_ptr;
.external   i_delay_line;
.external   i_delay_ptr;
.external   i_value;
.external   _value;
.entry      IQ;
IQ:
   {Look up Delta_phi value and add to phi for new phase          }
            ax1 = si;
            ay0 = 0x07;
            ar = ax1 and ay0;
            I7 = ^delta_phi;
            m7 = ar;
            modify(i7, m7);
            ax0 = dm(phi_value);
            ay0 = pm(i7,m4);
            ar = ax0 + ay0;
            ay1 = 0x0f;
            ar = ar and ay1;
            dm(phi_value) = ar;
```

(listing continues on next page)

```
   {Look up I and Q components                                           }
   {To find I/Q values, use phi/2 as offset into table                   }
         sr = lshift ar by -1 (lo);
   {Look at Amplitude bit and chose lookup table                         }
         ay0 = 0x08;
         ar = ax1 and ay0;
         if eq jump zamp;            {if zero amplitude bit is zero }
nzamp:   I6 = ^I_table_1;            {amplitude bit is equal to one }
         I7 = ^Q_table_1;
         jump lookup;
zamp:       I6 = ^I_table_0;         {amplitude bit is equal to zero}
         I7 = ^Q_table_0;
lookup:
         m7 = sr0;
         modify(I6,M7);              {I6 = I Value                  }
         modify(I7,M7);              {I7 = Q Value                  }
         mx0 = pm(i6,m4);
         mx1 = pm(i7,m4);
   {Write new I value to PSF Delay Line                                  }
         I6 = dm(i_delay_ptr);
         L6 = %i_delay_line;
         M6 = -1;
         dm(i6,m6) = mx0;
         dm(i_delay_ptr) = i6;
         L6 = 0;

         {Write IQ values for DAC                                        }
         dm(i_value) = mx1;
         dm(q_value) = mx0;
         rts;
.endmod;
```

Listing 2.32 IQ Generator Routine (IQ29.DSP)

2.8 REFERENCES

Bershad, J. N. 1989. "Nonlinear Quantization Effects in the LMS and Block LMS Adaptive Algorithms," *IEEE Trans. ASSP*, vol. 37, No. 10, pp.1504-1512.

Bingham A. J. 1988. *The Theory and Practice of Modem Design.* New York, NY: John Wiley & Sons.

Brady, D. M. 1970. "An Adaptive Coherent Diversity Receiver for Data Transmission through Dispersive Media," *IC Conference Record,* ICC 70 pp. 21-40.

CCITT, Eighth Plenary Assembly. 1985. *Red Book, Volume VIII, Fascicle VIII.1: Data Communication Over the Telephone Network.* Geneva: International Telecommunication Union.

Falconer, D. D. and L. Ljung. 1978. "Application of Fast Kalman Estimation to Adaptive Equalization," *IEEE Trans. Commun*., vol. COM-26, pp. 1439-1446.

Godard, D. 1974. "Channel Equalization Using a Kalman Filter for Fast Data Transmission," *IBM. J. Res. Develop*., vol. 18, pp. 267-273.

Kamilo, F. and D. Messerschmitt. 1987. *Advanced Digital Communications.* Englewood Cliffs, NJ: Prentice Hall.

Lee, Edward A. and David G. Messerschmitt. 1988. *Digital Communication*. Boston, MA: Kluwer Academic Publishers.

Lin, Shu and Daniel J. Costello, Jr. 1983. *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, N.J.: Prentice-Hall.

Long, G., Fuyun, L. and J. G. Proakis. 1989. "The LMS Algorithm with Delayed Coefficient Adaptation," *IEEE Trans. ASSP*, vol. 37, No. 9, pp. 1397-1405.

Lucky, R.W. 1965. "Automatic Equalization for Digital Communication," *Bell Syst. Tech. J.*, vol 44. pp. 547-588.

Proakis, J.G. and J. H. Miller. 1969. "An Adaptive Receiver for Digital Signaling through Channels with Intersymbol Interference," *IEEE Trans. Information Theory*, vol IT-15, pp. 484-497.

Proakis, John G. 1989. *Digital Communications*. Second Edition. New York, N.Y.: McGraw-Hill.

Proakis, John, G. and D.G. Manolakis. 1988. *Introduction to Digital Signal Processing*. New York, N.Y.: McMillan Publishing Co.

Quatieri, T. and G. O'Leary. 1989. "Far-Echo Cancellation in the Presence of Frequency Offset", *IEEE Transactions on Communications*. Volume 37, No. 6, pp. 635-634.

Satotius, E. H. and J. D. Pack. 1981. "Application of Least Squares Lattice Algorithms to Adaptive Equalization," *IEEE Trans. Commun.*, vol. COM-29, pp. 136-142.

Sklar, Bernard. 1988. *Digital Communications - Fundamentals and Applications*. Englewood Cliffs, N.J.: Prentice-Hall.

Ungerboeck, G. 1972. "Theory on the Speed of Convergence in Adaptive Equalizers for Digital Communication," *IBM. J. Res. Develop.*, vol. 16, pp. 546-555.

Ungerboeck, G. 1976. "Fractional Tap-spacing Equalizer and Consequences for Clock Recovery in Data Modems," *IEEE Trans. Commun.*, vol. COM-24, pp. 856-864.

Wang, J. D. and J. J. Werner. 1988. "Performance Analysis of an Echo Cancellation Arrangement that Compensates for Frequency Offset in the Far Echo", *IEEE Transactions on Communications*. Volume COM-36, No. 3, pp. 364-372.

Weinstein, S. 1977. "A Passband Data-Driven Echo Canceller for Full-Duplex Transmission on Two-Wire Circuits", *IEEE Transactions on Communications*. Volume COM-25, pp. 654-665.

Ziemer and Peterson. 1985. *Digital Communications and Spread Spectrum Systems*. New York: MacMillan Publications.