Application Note

Digital Echo Canceler and Complex Adaptive Equalizers Using the *WE*® DSP16/DSP16A Digital Signal Processor

Contents

Contributed by: *Shen Tu*

Introduction

Echo cancellation techniques have been widely used in baseband digital transmission systems to provide full duplex data traffic over a two-wire network. The implementation of an echo canceler using the *WE* DSP16/DSP16A Digital Signal Processor (DSP) is given in the first section of this application note. The DSP16/DSP16A is a general-purpose, 16-bit DSP. The echo canceler is based on an adaptive transversal filter using the least mean square (LMS) algorithm to update the filter coefficients. Two examples are given: one employs single-precision filter coefficients as well as single-precision algorithm calculations; the other uses double-precision arithmetic, except that the input and output data are represented as single-precision words. The trade-offs between the two systems are the data rate and echo rejection. The double-precision implementation can offer better echo rejection while the single-precision design allows a system with a higher data rate.

Complex adaptive equalizers are a widely used building block in digital communications systems. Their function is to equalize channel distortions in order to reduce intersymbol interferences (ISI) at the receiver. The second section of this application note describes complex adaptive equalizer implementations for both linear equalizers (LE) and decision feedback equalizers (DFE) using the DSP16/DSP16A. The LMS adaptation algorithm is employed to update the equalizer coefficients. Each equalizer is tested by applying a 16-point quadrature amplitude modulated (16-QAM) signal transmitted through a long-haul voiceband telephone channel. For an 8-tap LE, the DSP16/DSP16A program takes a maximum of 239 cycles to process each input data symbol. For a DFE consisting of a 3-tap forward equalizer and a 5-tap feedback equalizer, the program execution takes 265 cycles. Using a 55 ns DSP16/DSP16A processor, the throughputs for the

linear and decision feedback equalizers are under 14 μs and 15 μs, respectively.

Digital Echo Canceler Implementation

A two-wire, full duplex baseband data transmission system usually relies on an echo canceler to decouple the outgoing transmitted signal and the incoming received signal. An example of this system is the integrated service digital network's (ISDN) U interface which supports 160 Kbit/s (80K baud using 2B1Q – two binary to one quaternary – line code) data rate on a subscriber loop over distances up to 5 Km [1]*. The application of the echo canceler in systems of various data rates and distance coverages has also been reported [2,3]. In most systems, the worst case signal attenuation of 40—50 dB can be expected. Therefore, in order to achieve a signal-to-echo ratio of 20 dB as required for low bit error rate (BER) transmission, the echo canceler has to provide 60—70 dB echo rejection.

The echo canceler can be implemented as an adaptive transversal filter which automatically adapts its coefficients to the impulse response of the echo path. If the echo path is a linear system and the transversal filter contains sufficient taps to cover the entire period of the echo's impulse response, the echo replica generated by the transversal filter can completely cancel out the echo signal from the received signal. However, due to the finite precision of the DSP and the tap misadjustments inherent to the adaptation algorithm, a small residual echo always exists.

The LMS algorithm [4] is the most popular algorithm used in adaptive transversal filters. The computational requirements (measured by the multiplication/addition operation) is 2N+1 operations per data sample, where N is the number of taps. The implementation of the LMS adaptive filter using a fixed-point processor introduces two major sources of error: one is from the quantization error in the filter coefficients, and the other arises when the tap updates become smaller than the least significant digit of the processor. This causes the adaptation process to *stall*.

In this section, a 48-tap echo canceler implemented using both single-precision and double-precision arithmetic is described. Both implementations are tested using a simulated two-wire circuit which consists of a 22-gauge 10 Km loop, the termination network, and the transmit and receive filters. The results are compared with those obtained from using floating point arithmetic.

Digital Echo Canceler for Baseband Data Transmission

A two-wire data transmission system with echo cancellation is shown in Figure 1. Let the input signal sequence to the echo canceler be a(n), and the coefficients of the transversal filter be c(n). Then the echo replica at the output of the echo canceler is:

$$1)\quad \hat{e}(n) = \sum_{i=0}^{N-1} c_i(n)a(n-i).$$

The input to the receiver is the the far-end signal s(n) and the echo signal e(n). At the echo canceler output, the received signal is :

$$2)\quad r(n) = s(n) + e(n) - \hat{e}(n).$$

The mean square residual echo $E((e(n) - \hat{e}(n))^2)$ can be minimized by reducing the received signal $E(r^2(n))$; if s(n) and $e(n) - \hat{e}(n)$ are uncorrelated. Using the LMS algorithm, the coefficient adaptation follows:

$$3)\quad c_i(n+1) = c_i(n) + \mu r(n)a(n).$$

In order to achieve a high degree of cancellation and due to the far-end signal acting as noise to the adaptation process, the convergence factor μ must be set to a small value to reduce the coefficient misadjustment at steady-state. However, a small value of μ results in slow convergence. Also, as mentioned earlier, the echo canceler utilizing the adaptive transversal filter technique is only capable of canceling the linear echo term. The non-linear echo that might be introduced by a buffer amplifier, hybrid transformer, analog-to-digital converter, etc. will remain uncanceled.

Because of the timing requirements in the data transmission system, e.g., the ISDN, the echo canceler of data symbol rate is usually not adequate. Since the input rate is equal to symbol rate, this implies that the echo canceler should have a higher output rate than the input rate. Figure 2

* [] indicates a reference listed at the end of this section, pg. 6.

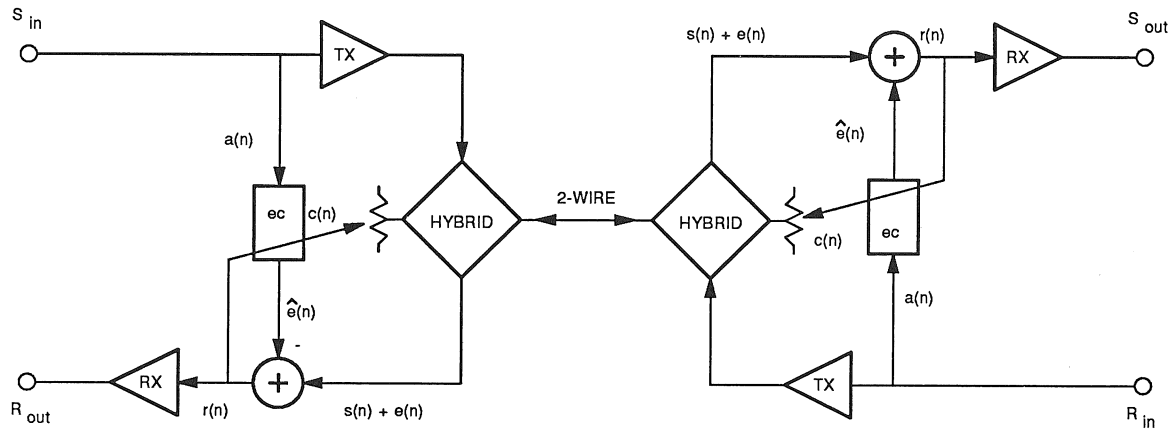
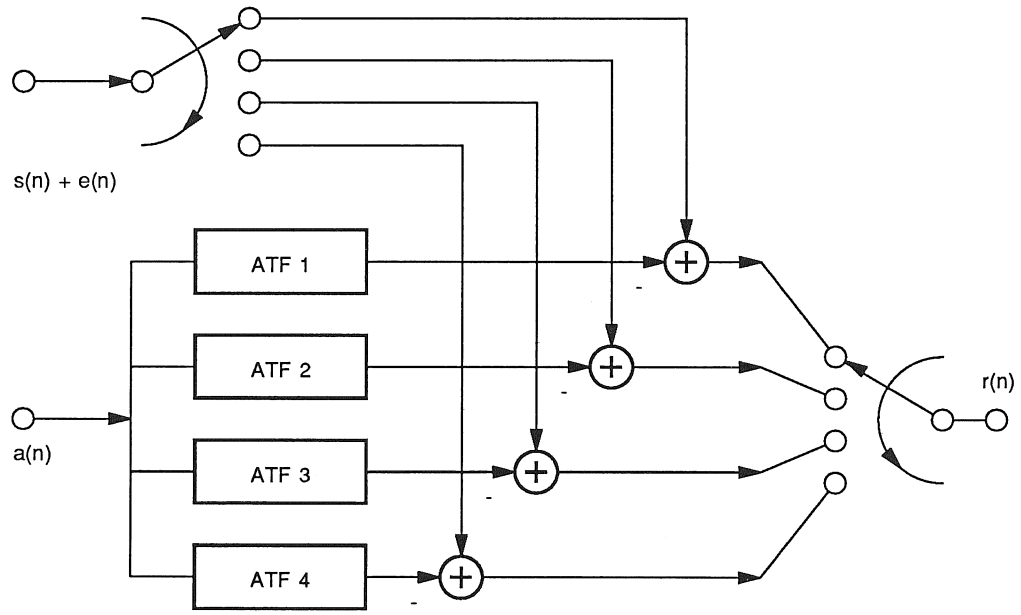


Figure 1. Diagram of Two-Wire Data Transmission System With Echo Canceler (ec)



ATF = Adaptive Transversal Filter.

Figure 2. Block Diagram of Echo Canceler With Interpolation Factor R = 4

shows the block diagram of an echo canceler with an interpolation factor of 4 (R=4). It consists of four independent transversal filters, each 12 taps. The input to the echo canceler is the transmitted data symbol sequence of rate 1/T. The output rate is 4/T, which corresponds to the echo replica of four samples in one symbol interval.

DSP16/DSP16A Implementation of the Echo Canceler

The implementation of the echo canceler using the DSP16/DSP16A processor involves programming the three operations described in equations (1)—(3) according to the block diagram in Figure 2. To perform the convolution of equation (1) using the DSP16/DSP16A, all multiplication products must be represented exactly using a 32-bit product register and rounding should be performed only after they are summed (in a 36-bit accumulator), i.e., at the filter output.* In this case, the output roundoff noise is uniformly distributed between $-Q/2$ and $Q/2$ with a mean of zero and a variance of $Q^2/12$, where Q is the quantization step size. The same roundoff noise is true for equation (2). In equation (3), if c is represented as a 16-bit word, any tap update smaller than the value rounded to a 16-bit word will be dropped, and the adaptation process cannot go any further to reach the minimum mean square error. In other words, adaptation stops when

4) least significant digit of the coefficient $\geq \mu r(n)a(n)$.

Therefore, in a fixed-point implementation of an adaptive transversal filter, there exists an optimum μ at which the inherent tap misadjustment is equal to the least significant digit of the coefficient.

If the level of cancellation cannot be met by single-precision arithmetic, double precision coefficients and coefficient update algorithm should be used instead. Using double-precision, the adaptation stopping criteria, equation (4), can be reduced by a factor of 2^{16}. However, the roundoff noise from the transversal filter should remain unchanged since the data in both cases is rounded off to 16-bit words. The DSP16/DSP16A is a 16-bit device which features a 16-bit data bus, a 16×16 2's complement multiplier that generates a full 32-bit product, and two 36-bit accumulators. To use double-precision (32-bit) coefficients, the 32-bit word must be broken into two 16-bit words for storage as well as arithmetic operations. One way to do this is to represent a 32-bit number in accumulator **a0** as

5) $\mathbf{a0} = a0_{MSW} + a0_{LSW}\, 2^{-16}$

where $a0_{MSW}$ is the 2's complement most significant word and $a0_{LSW}$ is the 2's complement least significant word of **a0** defined by the following equations:

6) $a0_{MSW}$ = rnd(a0);
rnd() = DSP16/DSP16A rounding instruction

7) $a0_{LSW} = a0l\, 2^{16}$;
a0l=lower half of a 32–bit word

This method utilizes three DSP16/DSP16A special function instructions: rounding, right shift 16 bits, and left shift 16 bits. There is no need to perform software unsigned multiplication that may be required using other approaches.

The program listings of echo cancelers using both single-precision and double-precision arithmetic are at the end of this section.

Simulation Results of the Echo Cancelers

The echo canceler programs are tested in the *env16* (an integrated DSP16/DSP16A assembler and simulator) environment. The input sequence to the echo canceler is a 19.2 Kbit/s random data sequence using bipolar coding. The echo path impulse response is shown in Figure 3, which corresponds to a 22-gauge, 10 Km loop with termination network and the transmit/receive filters. The far-end impulse response is shown in Figure 4. The echo canceler used in this case has a total of 48 taps and an interpolation factor R=4. The maximum impulse response duration is 12 input symbol intervals. The results of single-precision,

* Rounding instead of truncation should be performed in equations (1)—(3) in order to obtain the zero mean statistics; also, the bias as a result of numerical truncation in the adaptive filter coefficients can seriously degrade filter performance.

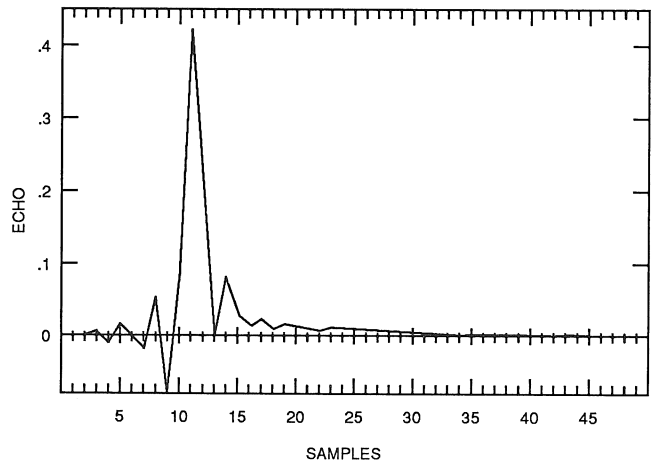


Figure 3. Echo Path Impulse Response

double-precision, and floating-point* implementations of the same echo canceler are listed in Table 1.

Table 1. Signal to Echo Ratio (dB) vs Iterations

No. of Symbols	Floating-Point	Single-Precision	Double-Precision
	μ=0.016	μ=0.016	μ=0.016
0	−29.06	−29.06	−29.06
2000	−9.79	−9.82	−9.79
4000	2.43	1.90	2.43
6000	4.81	4.00	4.81
8000	5.40	4.67	5.40
10000	5.65	5.09	5.66
12000	5.70	5.01	5.71
	μ=0.00025	μ=0.01	μ=0.00025
14000	11.46	7.09	11.40
16000	18.69	7.27	18.56
18000	20.04	7.21	20.01

At the start of adaptation, all coefficients are set to zero. The initial μ is chosen to be 0.016 for fast convergence, but the steady-state signal-to-echo ratio is only 5.7 dB. By using a smaller μ to continue the adaptation process, both floating-point and

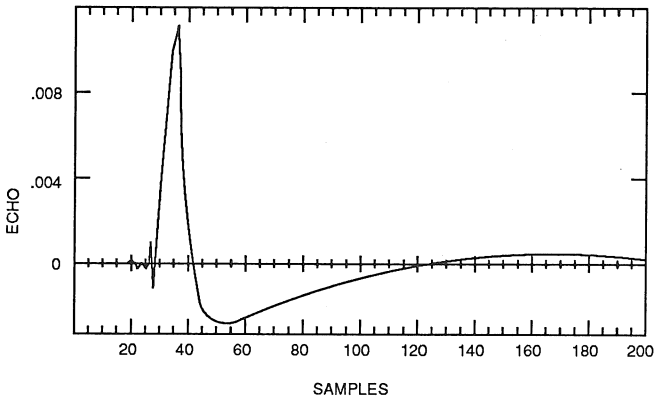


Figure 4. Far-End Impulse Response

double-precision implementations achieved the 20 dB signal-to-echo ratio. Notice that the results between the floating-point and double-precision implementations are very close. The slight discrepancy can be attributed to the quantization noise after rounding the echo canceler output (r(n)) to a 16-bit word. By imposing the criteria described by equation (4) on μ, the minimum μ that can be used without making the adaptation process stop is 0.01 in the single precision implementation. The final echo level is 7.21 dB below the received far-end signal. This is also close to 7.77 dB obtained using floating-point arithmetic with the same μ.

The single-precision implementation of a 48-tap echo canceler takes 445 machine cycles while the same canceler with double-precision arithmetic takes 1065 cycles to process each input data symbol and output four samples of far-end signal without echo. For a 33 ns version DSP16A, the data output rate is 3.7 and 8.8 μs, respectively. The single-precision implementation requires 101 words of ROM and 59 words of RAM. On the other hand, 177 words of ROM and 107 words of RAM are needed using double-precision program.

* The floating-point results are obtained from a C program on a SUN 3 computer using 32-bit single-precision floating-point arithmetic.

Conclusion

The implementation of two digital echo cancelers for baseband data transmission using the DSP16/DSP16A DSP is described in this application note. The single-precision echo canceler has a throughput advantage and occupies less memory space than its double-precision counterpart. However, the double-precision echo canceler offers much greater degree of cancellation and is limited only by the 16-bit quantization noise. The speed and flexibility of the DSP16/DSP16A device lends itself to the implementation of echo cancelers in two-wire full duplex baseband data transmission systems.

References

[1] O. Agazzi, "Cable Transmission Techniques for ISDN: A Tutorial", *Proceedings of the II Congresso da Sociedade Brasileira de Microeletronica*, July, 1987, Sao Paulo, Brazil, pp. 623-632.

[2] T. Alvestad, T. J.-C. Eriksen, "Echo Canceler for Two-Wire Data Modems", *Electrical Communication*, Vol. 59, No. 3, 1985, pp. 333-337.

[3] N.A.M. Verhoeckx et al. "Digital Echo Cancellation for Baseband Data Transmission", *IEEE Trans. Acoust., Speech, Signal Processing*, Vol. ASSP-27, pp. 768-781, Dec. 1979.

[4] B. Widrow, S. D. Sterns,*Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

Program Listing 1. Echo Canceler – Single-Precision

```
/*                                                                    */
/*        Digital Echo Canceler for Two-Wire Data                     */
/*      Transmission - Single Precision Coefficients                  */
/*                                                                    */
/*  by S. Tu, AT&T Bell Laboratories, Allentown, PA  10/28/88         */
/*                                                                    */

 .ram
  dat1:   11*int        /* allocate 12 ram spaces for 12-tap         */
 dat12:      int        /* delay lines                               */
   cf1:   12*int        /* 12 coef. for sub-canceler 1               */
   cf2:   12*int        /* 12 coef. for sub-canceler 2               */
   cf3:   12*int        /* 12 coef. for sub-canceler 3               */
   cf4:   12*int        /* 12 coef. for sub-canceler 4               */
    rx:      int
 .endram

 start:  auc=0x02                          /* set alignment for Q14 format */
         pt=CF1                            /* pt - coef. in rom            */
         r2=cf1                            /* r2 - coef. in ram            */

  do 48  {                                 /* transfer start-up coef.      */
                        y=a0     x=*pt++    /* from rom to ram (y=a0 is a   */
         *r2++=x                           /* dummy fetch)                 */
         }

         r1=dat12
         r3=rx
         rb=dat1                           /* rb points to beginning, re   */
         re=dat12                          /* to end of modulo.            */
         j=-12
         i=0
         c0=0                              /* initialize counter c0 to 0   */

  loop:  *r3=sdx                           /* read s(n)+e(n) from SIO      */
```

```
        if c0lt goto cnvl                       /* read a(n) from PIO every     */
        *r1++=pdx0                              /* 4th loop.                    */
        c0=-3                                   /* set c0 to -3                 */
        r2=cf1
                                                /* start convolution routine    */
cnvl:   a0=p            x=*r2++                 /* load x, y and clear a0.      */
        a0=a0-p         y=*r1++
do 11   {                                       /* perform convolution to       */
                p=x*y   x=*r2++                 /* obtain echo estimate.        */
        a0=a0+p         y=*r1++
        }

                p=x*y
        a0=a0+p         *r2++j                  /* end of convolution routine   */

        y=*r3                                   /* subtract estimated echo      */
        a0=a0-y                                 /* from received signal then    */
        a0=-a0                                  /* output the result to SIO     */
        a0=rnd(a0)                              /* after rounding               */
        sdx=a0
                                                /* start tap adaptation routine */
 lms:                   y=a0     x=*pt++i       /* scale a0 by the factor μ     */
                p=x*y                           /* then load to y for tap       */
        a0=p            x=*r1++                 /* update calculation           */
        a0=rnd(a0)
        y=a0

do 11   {                                       /* update filter coefficients   */
                p=x*y   a0=*r2
        a0=a0+p         x=*r1++
        a0=rnd(a0)
        *r2++=a0
        }
                p=x*y   a0=*r2
        a0=a0+p
        a0=rnd(a0)
        *r2++=a0                                /* end of adaptation routine    */

 end:   goto loop

                        /* filter final coef. settings */
 CF1:   int     0.00043
        int     -0.00098
        int     0.00177
        int     0.00360
        int     0.00592
        int     0.00879
        int     -0.00037
        int     0.14478
        int     0.01013
        int     -0.00085
        int     -0.00079
        int     0.0
        int     0.00055
        int     -0.00220
        int     0.00153
        int     0.00323
        int     0.00500
        int     0.00983
        int     0.02832
        int     0.30933
        int     0.01343
        int     -0.00128
        int     -0.00110
        int     0.00043
        int     0.00055
```

```
          int     -0.00189
          int     0.00116
          int     0.00269
          int     0.00421
          int     0.00885
          int     0.03125
          int     0.24213
          int     -0.03455
          int     -0.00446
          int     -0.00098
          int     0.00031
          int     0.00012
          int     -0.00049
          int     0.00055
          int     0.00256
          int     0.00378
          int     0.00757
          int     0.01325
          int     0.06293
          int     -0.02692
          int     -0.00574
          int     0.0
          int     0.00037
     U:   int     0.01
```

Program Listing 2. Echo Canceler – Double Precision

```
/*                                                              */
/*         Digital Echo Canceler for Two-Wire Data              */
/*       Transmission - Double Precision Coefficients           */
/*                                                              */
/*  by S. Tu, AT&T Bell Laboratories, Allentown, PA  11/28/88   */
/*                                                              */

 .ram
 dat1:   11*int          /* allocate 12 ram spaces for 12-tap    */
dat12:      int          /* delay lines                          */
  cf1:   24*int          /* 12 coef. for sub-canceler 1          */
  cf2:   24*int          /* 12 coef. for sub-canceler 2          */
  cf3:   24*int          /* 12 coef. for sub-canceler 3          */
  cf4:   24*int          /* 12 coef. for sub-canceler 4          */
  scl:    2*int
   rx:      int
 .endram

start:  auc=0x02                          /* set alignment for Q14 format  */
        pt=CF1                            /* pt - rom coef. pointer        */
        r2=cf1                            /* r2 - ram coef. pointer        */

 do 96  {                                 /* transfer start-up coef.       */
                      y=a0     x=*pt++     /* from rom to ram (y=a0 is a    */
          *r2++=x                         /* dummy fetch)                  */
        }

        r0=scl
        r1=dat12
        r3=rx
        rb=dat1                           /* rb points to beginning, re    */
        re=dat12                          /* to end of modulo.             */
        j=-24
        i=0
        c0=0                              /* initialize counter c0 to 0    */

 loop:  *r3=sdx                           /* read s(n)+e(n) from SIO       */
```

```
        if c0lt goto cnvl                       /* read a(n) from PIO every      */
        *r1++=pdx0                              /* 4th loop iteration.          */
        c0=-3                                   /* set c0 to -3                 */
        r2=cf1
                                                /* start convolution routine    */
cnvl:   a0=p            x=*r2++                 /* load x, y and clear a0,a1    */
        a0=a0-p         y=*r1++
        a1=a0

  do 11 {                                       /* perform double precision     */
              p=x*y     x=*r2++                 /* convolution to obtain echo   */
        a0=a0+p p=x*y   y=*r1++                 /* estimate                     */
        a1=a1+p         x=*r2++
        }
              p=x*y     x=*r2++
        a0=a0+p p=x*y                           /* a0- convolution output (LSW) */
        a1=a1+p         x=*r2++j                /* a1- convolution output (MSW) */
                                                /* end of convolution routine   */

        a0=a0>>16                               /* align a0 with a1             */
        y=a1                                    /* transfer a1h to yh first,    */
        a1=a1<<16                               /* then a1l to yl.              */
        yl=a1
        a0=a0+y         y=*r3                   /* add a0 and a1 to yield echo  */
                                                /* estimate; fetch s(n)+e(n)    */
        a0=a0-y                                 /* subtract estimated echo      */
        a0=-a0                                  /* from received signal, then   */
        a1=rnd(a0)                              /* output result to SIO after   */
        sdx=a1                                  /* rounding                     */

                        y=a1    x=*pt++i        /* fetch canceler output to y,  */
                                                /* convergence factor to x      */

                                                /* start lms adaptation routine */
  lms:          p=x*y   x=*r1++                 /* multiply x and y to yield 32 */
        a0=p                                    /* bit product;convert p to two */
        *r0++=a0l                               /* 16 bit words and store in    */
        a0=rnd(a0)                              /* ram location pointed to by r0*/
        *r0--=a0
   move y=*r0++

 do 11  {                                       /* update filter coefficients   */
              p=x*y     y=*r0--                 /* perform 32 x 16 -> 32        */
        a0=p  p=x*y     x=*r1++                 /* multiplication to yield 32   */
        a0=a0>>16                               /* bit coefficient update       */
        a0=a0+p         y=*r2++

        a1=rnd(a0)                              /* a1 - tap update (MSW)        */
        a0=a0<<16                               /* a0 - tap update (LSW)        */

        a0=a0+y         y=*r2--                 /* update the LSW of the coef.  */
        a0=a0>>16

        a1=a1+y         *r2++=a0l               /* update the MSW of the coef.  */
        a0=rnd(a0)
        y=a0
        a1=a1+y         y=*r0++
        *r2++=a1
        }

              p=x*y     y=*r0--
        a0=p  p=x*y
        a0=a0>>16
        a0=a0+p         y=*r2++

        a1=rnd(a0)
        a0=a0<<16
```

```
        a0=a0+y          y=*r2--
        a0=a0>>16

        a1=a1+y          *r2++=a0l
        a0=rnd(a0)
        y=a0
        a1=a1+y
        *r2++=a1                              /* end of adaptation routine   */

end:    goto loop
                           /* filter final coef. settings */
CF1:    int     0.83484
        int     -0.00024
        int     -1.17218
        int     0.00037
        int     -1.12122
        int     0.00177
        int     -0.19501
        int     0.00336
        int     0.58185
        int     0.00610
        int     -1.71277
        int     0.00873
        int     0.93817
        int     0.00012
        int     -0.39166
        int     0.14514
        int     -1.84900
        int     0.01007
        int     -1.12067
        int     -0.00043
        int     -1.42810
        int     -0.00055
        int     -1.91608
        int     -0.00024
        int     -0.77722
        int     -0.00055
        int     -1.00397
        int     -0.00024
        int     1.43420
        int     0.00104
        int     -1.47412
        int     0.00256
        int     1.48413
        int     0.00464
        int     -1.12616
        int     0.00934
        int     -0.84863
        int     0.02875
        int     -1.06116
        int     0.30914
        int     1.70483
        int     0.01263
        int     0.95929
        int     -0.00055
        int     0.95587
        int     -0.00092
        int     1.71930
        int     -0.00037
        int     -0.64282
        int     -0.00049
        int     0.57837
        int     -0.00024
        int     -0.49121
        int     0.00092
        int     0.26746
```

```
        int     0.00220
        int     -0.26996
        int     0.00421
        int     0.10040
        int     0.00818
        int     -0.78595
        int     0.03174
        int     -0.57611
        int     0.24237
        int     -0.25562
        int     -0.03516
        int     1.80664
        int     -0.00372
        int     -1.50757
        int     -0.00055
        int     0.93994
        int     -0.00031
        int     -1.84192
        int     -0.00012
        int     -0.39398
        int     0.
        int     1.40552
        int     0.00092
        int     0.29498
        int     0.00226
        int     1.03375
        int     0.00403
        int     1.17072
        int     0.00757
        int     -1.82428
        int     0.01349
        int     -0.10437
        int     0.06342
        int     -0.08191
        int     -0.02716
        int     0.14935
        int     -0.00543
        int     1.98663
        int     0.
        int     -0.93744
        int     -0.00018
U:      int     0.00025
```

Complex Adaptive Equalizers

Transmitting high bit-rate data successfully through a switched telephone network, microwave or satellite link, etc., often requires adaptive equalization to remove excessive channel distortions (i.e., amplitude and phase distortion or multipath fading distortion). The adaptive equalizer is usually implemented using transversal filters with certain algorithms controlling the update of the coefficients [1].* The two most popular structures used in adaptive equalization are the LE and DFE.

The LE, shown in Figures 5 and 6, derives its output by linearly combining the past received signals weighted by the coefficients. The DFE, shown in Figures 7 and 8, is comprised of both a forward equalizer and a feedback equalizer. The output of the DFE is the sum of the two equalizers. The DFE is found to be particularly effective when the channel is highly distorted. In addition, it offers the advantages of being less sensitive to symbol timing accuracy and having less noise enhancement problems, as compared to linear equalizers.

* [] indicates a reference listed at the end of this section, pg. 16.

In this section, the implementations of an 8-tap complex linear equalizer and a DFE of 3 forward taps and 5 feedback taps using the DSP16/DSP16A processor are presented. Included is a description of the system blocks and the algorithm, the DSP16/DSP16A programs, and the testing results of the equalization of a 16-QAM signal sent through a dispersive voiceband telephone channel.

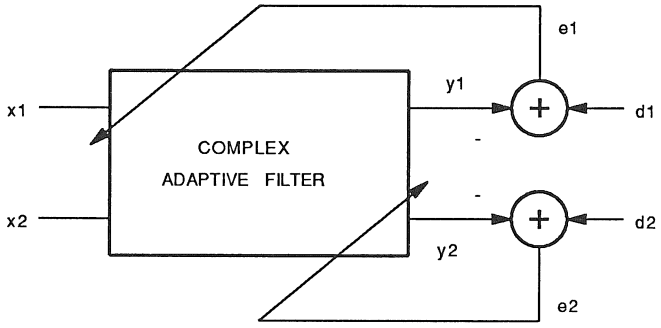


Figure 5. Complex Linear Equalizer Block Diagram

System Description and Algorithm

Figure 5 shows the block diagram of a complex adaptive equalizer. Inputs x1 and x2 correspond to the real and imaginary data paths. Outputs y1 and y2 of the complex transversal filter (Figure 6) are given in the following equations:

$$8)\quad y1(n) = \sum_{i=0}^{N-1} c1_i(n)x1(n-i) - \sum_{i=0}^{N-1} c2_i(n)x2(n-i);$$

N = number of taps

$$9)\quad y2(n) = \sum_{i=0}^{N-1} c2_i(n)x1(n-i) + \sum_{i=0}^{N-1} c1_i(n)x2(n-i)$$

The well-known LMS algorithm [2] is used to update coefficients c1 and c2 to minimize the mean square error, $E[\, e1^2 + e2^2\,]$. With a quadratic error function, the minimum mean square error is obtained where

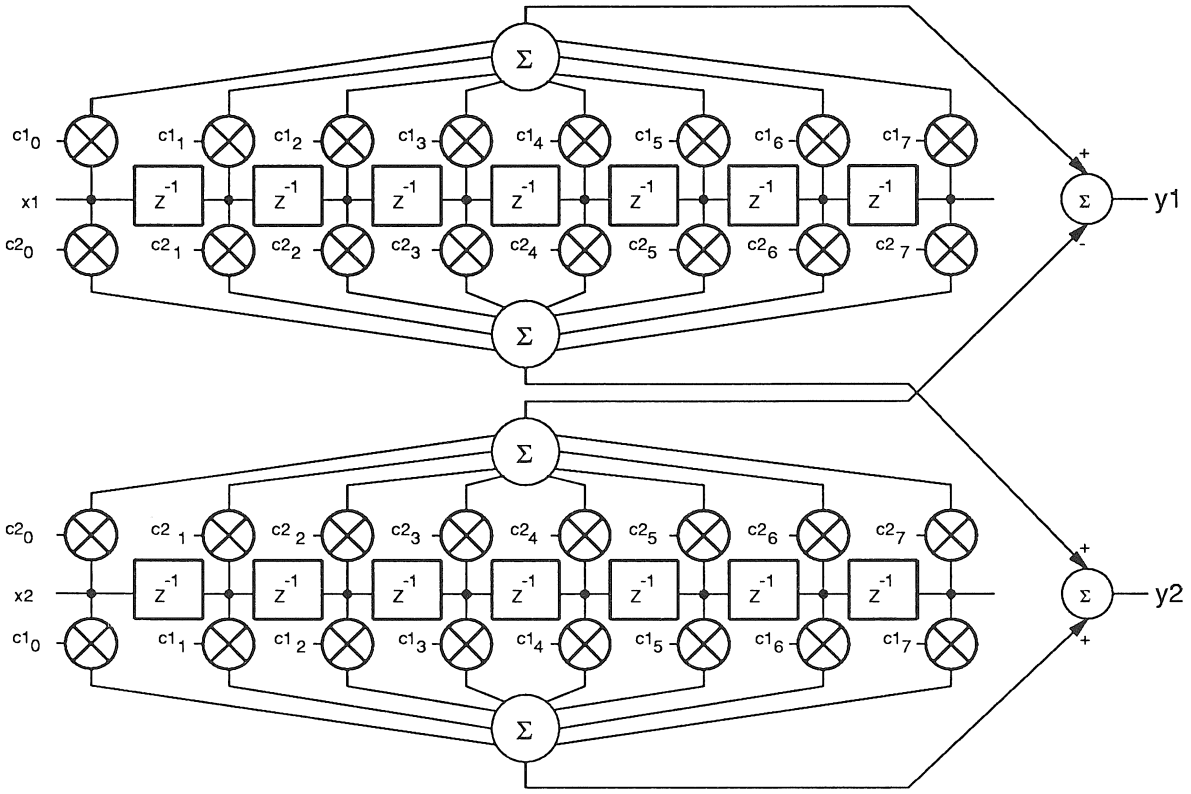


Figure 6. Complex Transversal Filter

the gradient is zero. The estimated gradient for c1 is e1x1 + e2x2 and for c2 is e2x1 – e1x2. The coefficient adaptation follows

10) $c1_i(n+1)=c1_i(n)+mu[e1(n)x1(n-i)+e2(n)x2(n-i)]$

11) $c2_i(n+1)=c2_i(n)+mu[e2(n)x1(n-i)-e1(n)x2(n-i)]$.

Error signals e1 and e2 are derived from the difference between the filter outputs and the desired outputs (d1 and d2). When the filter is used as a data equalizer, d1 and d2 are obtained from the quantized value of y1 and y2. In the case of a 16-QAM system, d1 and d2 can be represented by 2 bits. Therefore, a 2-bit quantizer is employed. The parameter μ is the convergence factor which determines the speed of convergence and the misadjustment of adaptation process.

Figures 7 and 8 show block diagrams of a complex adaptive DFE which consists of forward and feedback equalizers. The forward equalizer is used to equalize the pre-cursor ISI, while the feedback equalizer cancels the post-cursor ISI. The outputs, which correspond to the real and imaginary data paths of the DFE, are **z1 = f1 + y1** and **z2 = f2 + y2**. The forward equalizer outputs, y1 and y2, and the tap update equations are the same as those for the LE. The feedback equalizer outputs, f1 and f2, and coefficient adjustments are given in the following equations:

12) $$f1(n) = \sum_{i=1}^{M} b1_i(n)d1(n-i) - \sum_{i=1}^{M} b2_i(n)d2(n-i);$$

M = number of taps

13) $$f2(n) = \sum_{i=1}^{M} b2_i(n)d1(n-i) + \sum_{i=1}^{M} b1_i(n)d2(n-i)$$

14) $b1_i(n+1) = b1_i(n) + \mu[e1(n)d1(n-i) + e2(n)d2(n-i)]$

15) $b2_i(n+1) = b2_i(n) + \mu[e2(n)d1(n-i) - e1(n)d2(n-i)]$

All coefficients in the forward and feedback equalizer are adjusted simultaneously using the LMS algorithm to minimize the mean square error, $E[e1^2 + e2^2]$.

One problem unique to the DFE occurs when either d1 or d2 is detected incorrectly. In this case, the erroneous data stays in the feedback equalizer for several symbol periods, depending on the length of the tap. As a result, the incorrect data may cause successive errors. This effect is known as *error propagation.*

DSP16/DSP16A Implementation of Complex Adaptive Equalizers

The implementation of either the LE or DFE using the DSP16/DSP16A processor involves programming the convolution and tap adaptation algorithms described by equations (7)—(15) and the quantizer for obtaining signals d1 and d2. Before the program enters into the main loop, the initialization procedure transfers the start-up coefficient settings and the constants used by the quantizer from ROM to RAM. Although the constants can be left in ROM, they can be fetched in one cycle when they are stored in RAM, whether the instructions used to fetch the constants are executed from within the cache or not. The data is input and output via the serial I/O (SIO) port. Real and imaginary data inputs are transferred from SIO register **sdx** to data memory and then moved to the tapped delay line before each new iteration begins. Since the complex adaptive equalizer requires two consecutive input/output operations for synchronous data transfer, the **sdx** read/write commands are carefully placed within the program to insure that all data is shifted in/out before the next I/O. Asynchronous I/O can also be used by adding a simple polling routine before the **sdx** read/write command. (Refer to the *WE® DSP16 and DSP16A Digital Signal Processors Information Manual* for an example of a polling routine.)

In the LE, the convolution calculation of both data paths, described by equations (8) and (9), is executed from the cache with data and coefficients in RAM. The data memory for the tapped delay line is arranged as follows: x1(0), x2(0), x1(–1), x2(–1), ... x1(–7), x2(–7). The coefficients are arranged in memory the same way. The data memory organization is shown in Figure 9. The modulo addressing capability of the DSP16/DSP16A is most suitable for convolution computation and, therefore, is used in performing this function. The error used in the coefficient update calculation is the difference between the input and output of the quantizer. The calculation of coefficients follows equations (10) and (11). The negative convergence factor, $-\mu$, is stored in ROM, and its value is determined by the system parameters (i.e., the number of equalizer taps, the characteristics of the channel, etc.).

The difference between the LE and DFE is the addition of the feedback equalizer in the DFE. Therefore, the programming of the DFE involves switching tasks between the forward and feedback

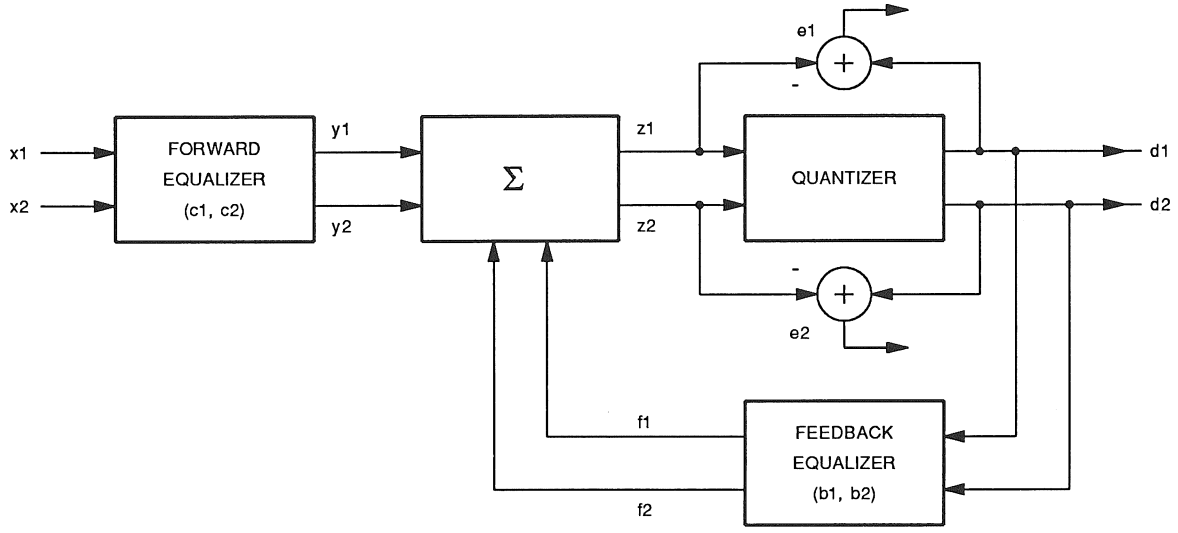


Figure 7. Complex Decision Feedback Equalizer Block Diagram

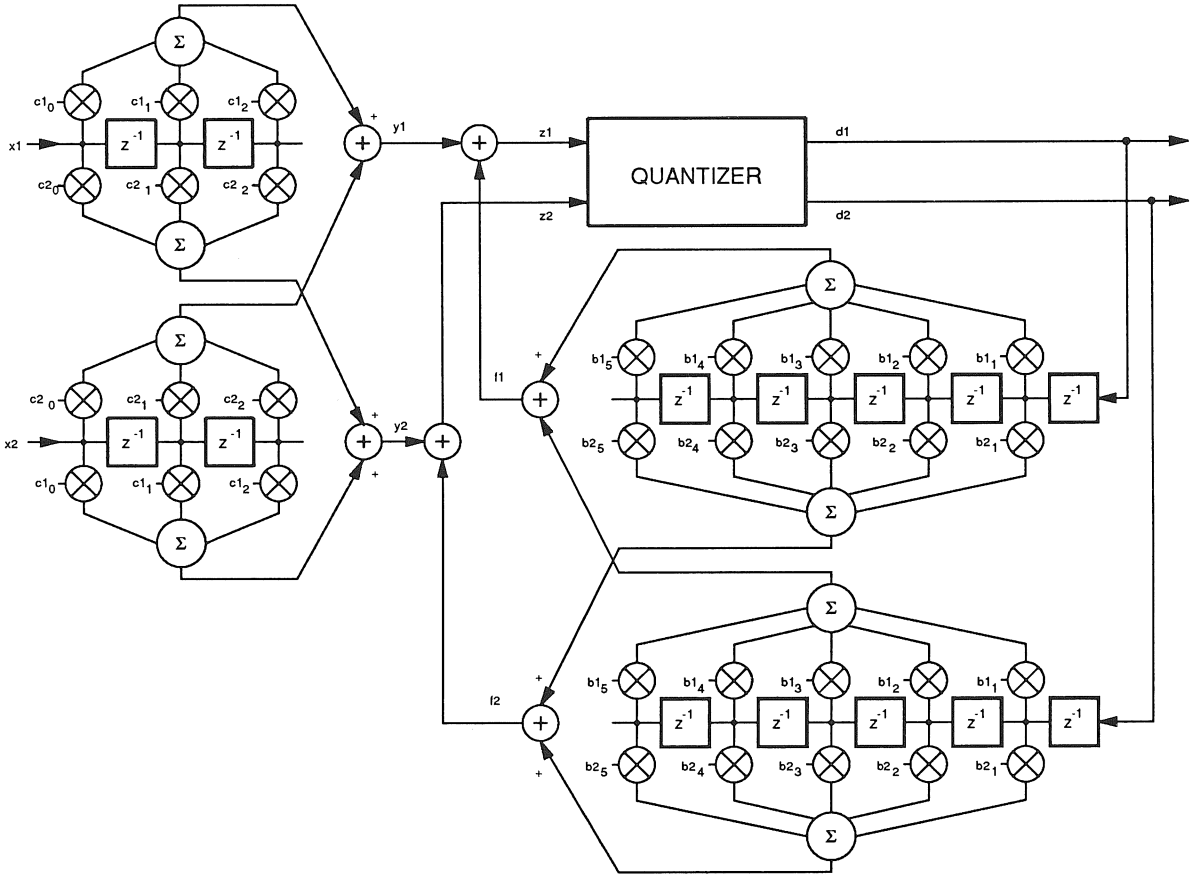


Figure 8. Complex Transversal Filters in the DFE

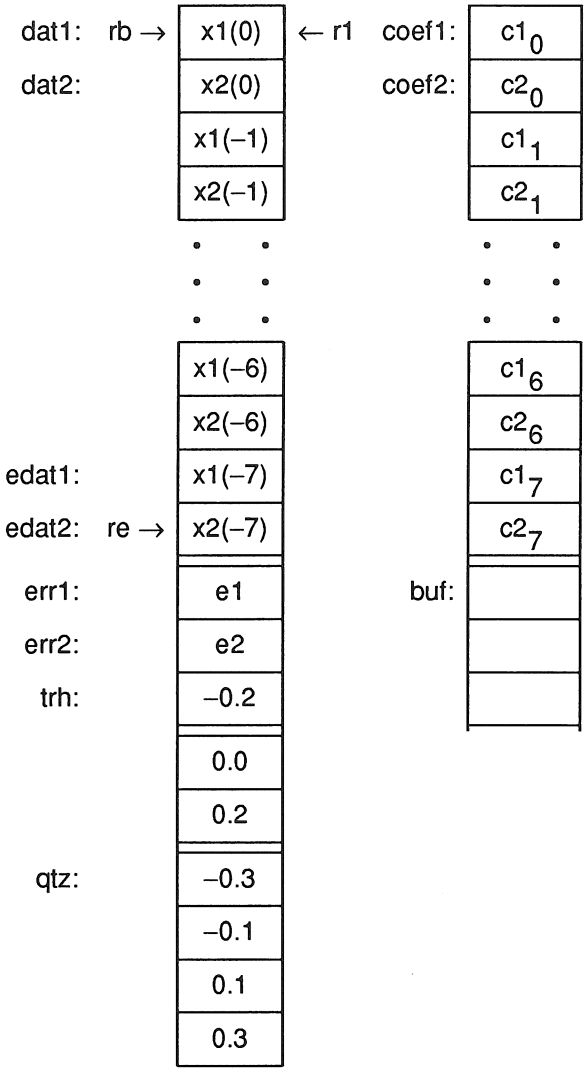


Figure 9. Data Memory Organization of Linear Equalizer

equalizers. Pointers **rb, re** and **r1** are used to configure the forward tapped delay line and must be saved for the next data input. The new pointer settings must be transferred to these registers before the feedback equalizer can be computed. Using the compound addressing feature provided by DSP16/DSP16A, such memory and register content swaps can easily be accomplished. The data memory organization of the DFE is shown in Figure 10.

The 2-bit quantizer is implemented as a subroutine which can be called to obtain both signals d1 and d2. Also, it can easily be replaced by a different subroutine (i.e., 3-bit quantizer in a 64-QAM case) without altering the main program. The complete program listings for the LE and DFE are given in Programs 3 and 4, respectively.

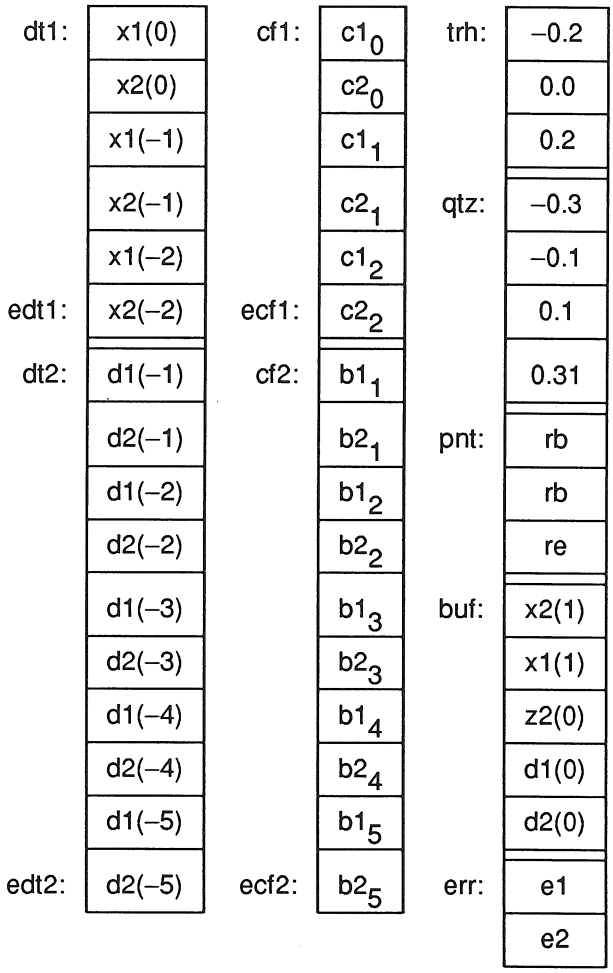


Figure 10. Data Memory Organization of Decision Feedback Equalizer

Testing of the Equalizer

A computer-generated test signal was applied to the input of the equalizer. A CCITT V.22 bis-compatible signal, which is a 16-point QAM signal, was used in this example. Its baseband pulse was shaped to a 75% raised cosine frequency spectrum. Figure 11 shows the eye-pattern plot of the baseband signal. The modulating carrier is 2400 Hz, which allows the transmission of 600 baud data through the higher frequency band of the voiceband telephone channel. The simulated channel represents the worst-case long-haul connection, according to the delay and amplitude distortion data published by a 1969 connection survey [3].

The LE and DFE programs were run under the *env16* environment (an integrated DSP16/DSP16A

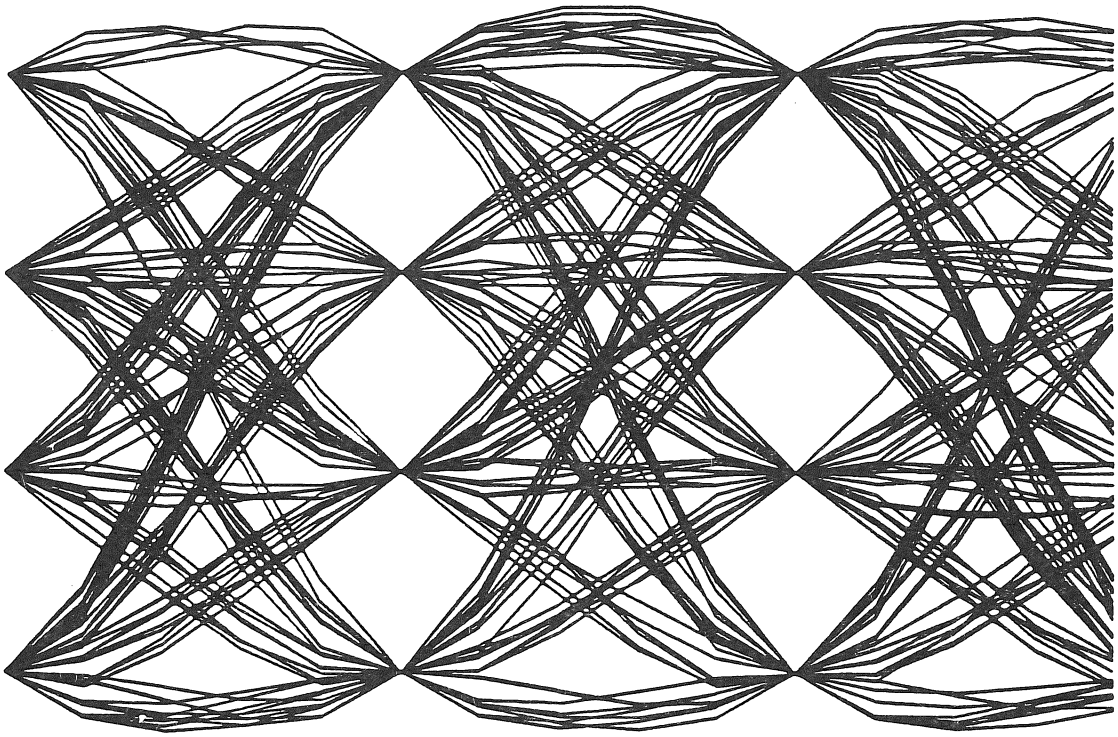


Figure 11. Eye Pattern of Baseband Data Signal

assembler and simulator) to obtain the following results. Figures 12 and 13 show the learning curves of the adaptation process in which the ratio of error power, $e1^2 + e2^2$, and the average transmitted signal power are plotted against the number of iterations. The constellation plots of the input and output of the equalizer (Figures 14—18) demonstrate the effectiveness of the equalizer in improving the ISI caused by channel distortions.

For the LE, there are a maximum of 239 cycles required to process each data point and 85 cycles to initialize the program. The entire program occupies 132 ROM words and 43 RAM words. The decision feedback equalizer takes slightly more execution cycles (265) and memory spaces (178 ROM words and 48 RAM words).

Conclusion

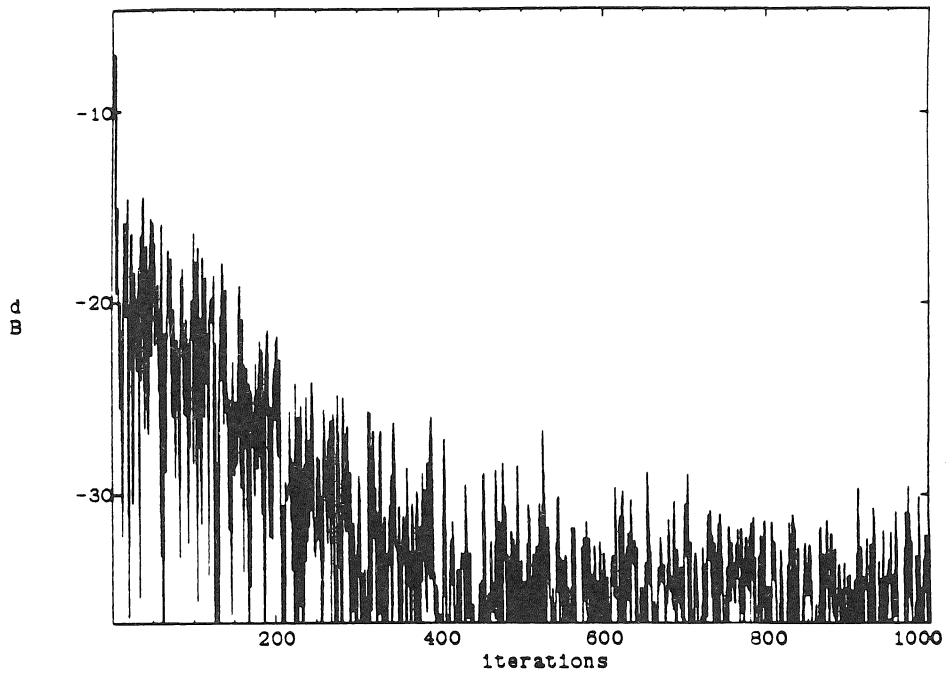
Implementations of an 8-tap complex linear equalizer and a decision feedback equalizer of 3 forward taps and 5 feedback taps using the DSP16/DSP16A DSP are given in this application note. Using a 55 ns processor, the throughputs of the LE and DFE are under 14 μs and 15 μs, respectively. The misadjustments of both adaptive equalizers at steady state are very low even though the full dynamic range ($-2 \leq X \leq 2-2^{-14}$ of Q14 format; signal range is $-0.4 < X < 0.4$) is not utilized. This low level of misadjustment is achieved by taking advantage of the two 36-bit accumulators and by using rounding instead of truncation (default operation) prior to the transfer of data from an accumulator to data memory or other 16-bit registers. The execution of the LMS complex adaptive filter alone takes 17 cycles per tap including rounding the new coefficient and writing it to the data memory.

References

[1] S. U. H. Qureshi, "Adaptive Equalization" *Proceedings IEEE*, Vol. 73, No.9, Sept. 1985, pp. 1349-1387.

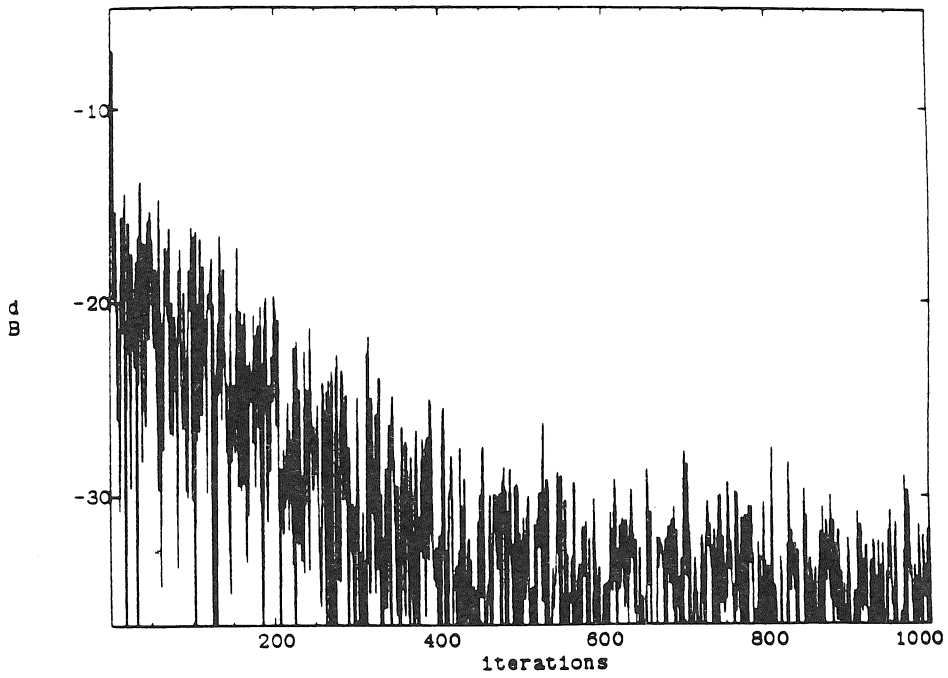
[2] B. Widrow, S. D. Sterns, *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[3] F. P. Duffy, T. W. Thatcher, "Analog Transmission Performance on the Switched Telecommunications Network" *BSTJ*, Vol. 50, No. 4. Apr. 1971, pp. 1311-1346.



Error-power to average-transmitted-power ratio vs number of iterations.

Figure 12. Learning Curve (LE)



Error-power to average-transmitted-power ratio vs number of iterations.

Figure 13. Learning Curve (DFE)

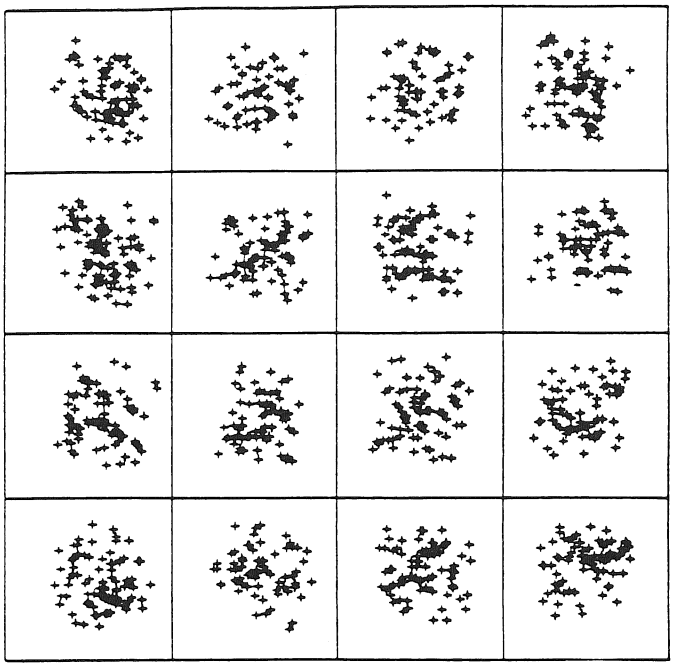


Figure 14. Constellation Plot of Input Data to Adaptive Equalizer (1200 samples)

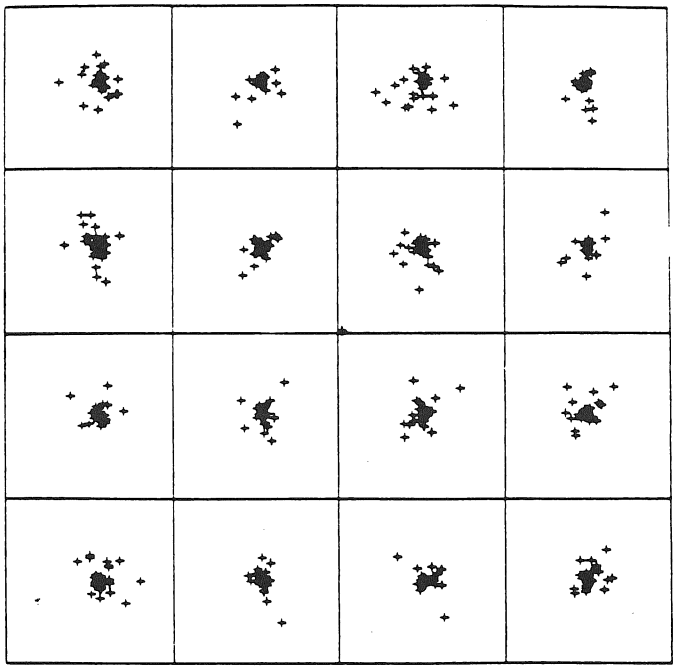


Figure 15. Constellation Plot of LE Output (data 1—600)

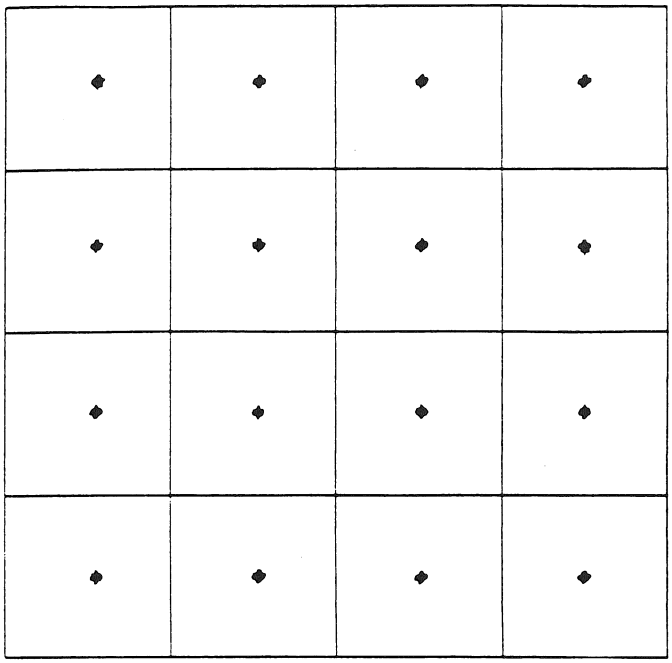


Figure 16. Constellation Plot of LE Output (data 601—1200)

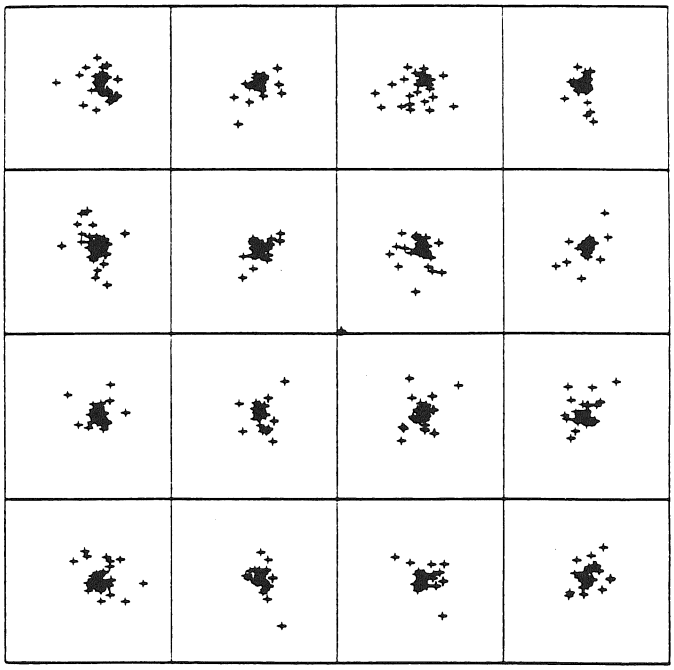


Figure 17. Constellation Plot of DFE Output (data 1—600)

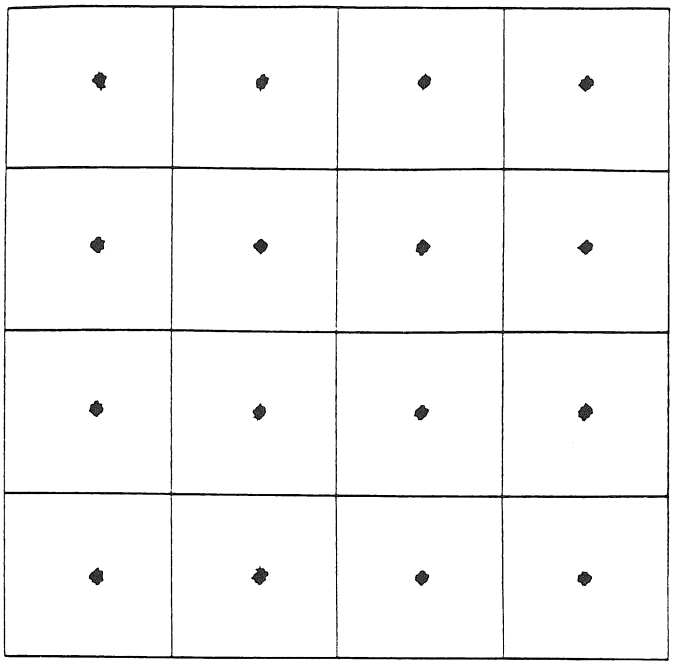


Figure 18. Constellation Plot of DFE Output (data 601—1200)

Program Listing 3. Complex Linear Equalizer

```
/*                                                                  */
/*              Complex Linear Equalizer                rev. 1.0    */
/*  by S. Tu, AT&T Bell Laboratories, Allentown, PA     12/20/88    */

 .ram
 dat1:        int     /* allocate 16 ram spaces for*/
 dat2:     13*int     /* 8-tap complex data        */
edat1:        int
edat2:        int
 err1:        int     /* err1 - real error signal   */
 err2:        int     /* err2 -imag. error signal   */
  trh:      3*int     /* decision threshold levels  */
  qtz:      4*int     /* quantized signal levels    */
coef1:        int     /* allocate 16 ram spaces for */
coef2:     13*int     /* 8-tap complex coefficients */
ecef1:        int
ecef2:        int
  buf:      3*int     /* temporary data storage     */
 .endram

start:  auc=0x02                          /* set alignment for Q14 format */
        pt=CI                             /* pt -> coef. in rom           */
        r2=coef1                          /* r2 -> coef. in ram           */
  do 16 {                                 /* transfer coefficients from   */
                    y=a0    x=*pt++       /* rom to ram (y=a0 is a dummy  */
        *r2++=x                           /* fetch)                       */
        }
        r0=trh                            /* r0 -> threshold in ram       */
   do 7 {                                 /* transfer threshold levels    */
                    y=a0    x=*pt++       /* and quantization levels      */
        *r0++=x                           /* from rom to ram              */
        }
```

```
        r1=dat1                                 /* r1 -> data pointer          */
        rb=dat1                                 /* set virtual shifting        */
        re=edat2                                /* registers rb, re            */
        j=2
        i=0

 loop:  r0=buf                                  /* main loop                   */
        *r0++=sdx                               /* input real data from SIO    */
        r2=coef1
        r3=coef2
                                                /* convolution                 */
cnvl:   a0=p            x=*r2                   /* x = c1                      */
        a0=a0-p         y=*r1++                 /* clear a0; y = x1            */
        a1=a0                                   /* clear a1                    */
                p=x*y   x=*r3++j                /* p = x1*c1; x = c2           */
   do 7 {
        a0=a0+p p=x*y   y=*r1++                 /* p = x1*c2; y = x2           */
        a1=a1+p p=x*y   x=*r2++j                /* p = x2*c2; x = c1           */
        a0=a0-p p=x*y   y=*r1++                 /* p = x2*c1; y = x1(-1)       */
        a1=a1+p         x=*r2                   /* x = c1_1                    */
                p=x*y   x=*r3++j                /* p = x1(-1)*c1_1; x = c2_1   */
        }
        a0=a0+p p=x*y   y=*r1++
        a1=a1+p p=x*y   x=*r2++j
        a0=a0-p p=x*y
        a1=a1+p

        a0=rnd(a0)                              /* roundoff equalizer output   */
        a1=rnd(a1)                              /* a0,a1 to 16-bit word        */

        *r0++=sdx                               /* input imag. data from SIO   */
        *r0=a1                                  /* store imag. output data     */
        sdx=a0                                  /* output real data to SIO     */

/* error signal calculation of 16-QAM case    */
        r3=err1                                 /* r3 -> error signal (real)   */
        call dec                                /* call subroutine dec         */
        y=*r2                                   /* fetch quantized level d1    */
        a0=a0-y                                 /* calculate -e1 = y1 - d1     */
                        y=a0    x=*pt++         /* fetch convergence factor u  */
                p=x*y                           /* calculate  e1 = -u*(-e1)    */
        a0=p
        a0=rnd(a0)
        *r3++=a0                                /* write  e1 to ram (err1)     */

        a0=a1                                   /* repeat for  e2 calculation  */
        call dec
        y=*r2
        a0=a0-y
                        y=a0    x=*pt++i
                p=x*y
        a0=p
        a0=rnd(a0)
        *r3=a0                                  /* write  e2 to ram (err2)     */
```

```
/* coefficient adaptation using the LMS algorithm  */
        r2=err1                                 /* r2 -> err1                  */
        r3=err2                                 /* r3 -> err2                  */
        r0=coef1                                /* r0 -> coef1                 */
lms:    x=*r2                                   /* x = e1                      */
        y=*r1++                                 /* y = x1                      */

   do 7 {
                p=x*y   x=*r3                   /* p = e1*x1; x = e2           */
        a0=p    p=x*y   y=*r1++                 /* p = e2*x1; y = x2           */
        a1=p    p=x*y   x=*r2                   /* p = e2*x2; x = e1           */
        a0=a0+p p=x*y   y=*r0++                 /* p = e1*x2; y = c1           */
        a0=rnd(a0)
        a0=a0+y         y=*r0--                 /* a0 = c1(n+1); y = c2        */
        a1=a1-p         *r0++=a0                /* update c1                   */
        a1=rnd(a1)
        a1=a1+y         y=*r1++                 /* a1 = c2(n+1); y = x1        */
        *r0++=a1                                /* update c2                   */
        }
                p=x*y   x=*r3
        a0=p    p=x*y   y=*r1--
        a1=p    p=x*y   x=*r2
        a0=a0+p p=x*y   y=*r0++
        a0=rnd(a0)
        a0=a0+y         y=*r0--
        a1=a1-p         *r0++=a0
        a1=rnd(a1)
        a1=a1+y
        *r0++=a1

        y=*r0++                                 /* transfer new data to delay  */
        *r1++=y                                 /* line                        */
        y=*r0++
        *r1--=y
        sdx=*r0                                 /* output imag. data to SIO    */
        goto loop                               /* return to main loop         */

/* subroutine: dec - 2-bit quantizer           */
dec:    r0=trh                                  /* r0 -> trh                   */
        r2=qtz                                  /* r2 -> qtz                   */
        y=*r0++                                 /* fetch threshold level -0.2  */
        a0-y                                    /* to y and compare with a0    */
        if le return                            /* if greater, increment r2    */

        *r2++
        y=*r0++                                 /* compare with threshold 0,   */
        a0-y                                    /* if greater,increment r2     */
        if le return

        *r2++
        y=*r0                                   /* compare with threshold 0.2  */
        a0-y                                    /* if greater, increment r2    */
        if le return

        *r2++
        return
```

```
CI:  int     0       /* start of real coef.       */
CQ:  int     0       /* start of imaginary coef. */
     int     0
     int     0
     int     0
     int     0
     int     1.0
     int     0
     int     0
     int     0
     int     0
     int     0
     int     0
     int     0
     int     0
     int     0
TRH: int    -0.2     /* threshold levels          */
     int     0.0
     int     0.2
QTZ: int    -0.3     /* quantization levels       */
     int    -0.1
     int     0.1
     int     0.3
U:   int    -0.0625 /* -( convergence factor )   */
```

Program Listing 4. Complex Decision Feedback Equalizer

```
/*                                                              */
/*        Complex Decision Feedback Equalizer       rev. 1.0   */
/*  by S. Tu, AT&T Bell Laboratories, Allentown, PA  1/10/89   */

 .ram
 dt1:     5*int     /* Allocate 6 RAM spaces for          */
edt1:       int     /* 3-tap forward equalizer.           */
 dt2:     9*int     /* Allocate 10 RAM spaces for         */
edt2:       int     /* 5-tap feedback equalizer.          */
 cf1:     5*int     /* Allocate 6 RAM spaces for          */
ecf1:       int     /* 3-tap forward equalizer coef.      */
 cf2:     9*int     /* Allocate 10 RAM spaces for         */
ecf2:       int     /* 5-tap feedback equalizer coef.     */
 trh:     3*int     /* decision threshold level           */
 qtz:     4*int     /* quantized signal level             */
 pnt:     3*int     /* pointer r1,rb and re               */
 buf:     5*int     /* temporary data storage buffer      */
 err:     2*int     /* error signal                       */
 .endram

start:  auc=0x02                          /* set alignment for Q14 format  */
        j=2
        i=0
        pt=CI                             /* pt -> coef. in ROM            */
        r0=cf1                            /* r2 -> coef. in RAM            */
  do 23 {                                 /* Transfer coefficients and     */
                      y=a0   x=*pt++      /* constants from ROM to RAM     */
        *r0++=x                           /* (y=a0 is a dummy fetch).      */
        }
        r1=dt2                            /* r1 -> data pointer            */
        rb=dt2                            /* Configure tapped delay line   */
        re=edt2                           /* using rb, re for feedback     */
        *r0++=r1                          /* equalizer, save pointers      */
        *r0++=rb                          /* in RAM.                       */
        *r0++j=re
```

```
        r1=dt1                                  /* Configure tapped delay line   */
        rb=dt1                                  /* for forward equalizer.        */
        re=edt1

                                                /* main loop                     */
 loop:  *r0=sdx                                 /* input real data from SIO      */
        r2=cf1                                  /* convolution - forward eqlzr   */
cnvl1:  a0=p            x=*r2++                 /* x = c1                        */
        a0=a0-p         y=*r1++                 /* clear a0; y=x1                */
        a1=a0                                   /* clear a1                      */
                p=x*y   x=*r2--                 /* p = x1*c1; x = c2             */
   do 2 {
        a0=a0+p p=x*y   y=*r1++                 /* p = x1*c2; y = x2             */
        a1=a1+p p=x*y   x=*r2++j                /* p = x2*c2; x = c1             */
        a0=a0-p p=x*y   y=*r1++                 /* p = x2*c1; y = x1(-1)         */
        a1=a1+p         x=*r2++                 /* x = c1_1                      */
                p=x*y   x=*r2--                 /* p = x1(-1)*c1_1; x = c2_1     */
        }

        a0=a0+p p=x*y   y=*r1++
        a1=a1+p p=x*y   x=*r2++j
        a0=a0-p p=x*y
        a1=a1+p         x=*r2++                 /* x = b1                        */

        r0=pnt
        *r0zp:r1                                /* Exchange rb, re, r1           */
        *r0zp:rb                                /* contents with data in RAM     */
        *r0zp:re                                /* for the feedback equalizer    */
                                                /* convolution.                  */
cnvl2:  y=*r1++                                 /* y = d1                        */
                p=x*y   x=*r2--                 /* p = d1*b1; x = b2             */
   do 4 {
        a0=a0+p p=x*y   y=*r1++                 /* p = d1*b2; y = d2             */
        a1=a1+p p=x*y   x=*r2++j                /* p = d2*b2; x = b1             */
        a0=a0-p p=x*y   y=*r1++                 /* p = d2*b1; y = d1(-1)         */
        a1=a1+p         x=*r2++                 /* x = b1_1                      */
                p=x*y   x=*r2--                 /* p = d1(-1)*b1_1; x = b2_1     */
        }
        a0=a0+p p=x*y   y=*r1++
        a1=a1+p p=x*y   x=*r2++j
        a0=a0-p p=x*y
        a1=a1+p

        a0=rnd(a0)                              /* Roundoff equalizer output     */
        a1=rnd(a1)                              /* a0, a1 to 16-bit word.        */

        *r0++j=sdx                              /* input imag. data from SIO     */
        sdx=a0                                  /* output real data to SIO       */
        *r0++=a1                                /* Save imag. output data        */
                                                /* for later writing to sdx.     */

/* error signal calculation for 16-QAM case   */
        call dec                                /* call subroutine dec           */
        y=*r2                                   /* fetch quantized level d1      */
        a0=a0-y         *r0++j=y                /* -e1 = y1-d1; save d1          */
                        y=a0    x=*pt++i        /* fetch convergence factor u    */
                p=x*y                           /* e1 = -u*(-e1)                 */
        a0=p
        a0=rnd(a0)                              /* round e1 to 16 bit            */
        *r0--=a0                                /* save e1 in ram (err1)         */

        a0=a1                                   /* repeat for e2 calculation     */
```

```
        call dec
        y=*r2
        a0=a0-y         *r0++j=y
                        y=a0    x=*pt++i
                p=x*y
        a0=p
        a0=rnd(a0)
        *r0--=a0

/* feedback equalizer coefficient adaptation using the LMS algorithm          */
        r2=err
        r3=cf2
lms1:   x=*r2++                                 /* x = e1                     */
        y=*r1++                                 /* y = d1                     */
   do 4 {
                p=x*y   x=*r2--                 /* p = e1*d1; x = e2          */
        a0=p    p=x*y   y=*r1++                 /* p = e2*d1; y = d2          */
        a1=p    p=x*y   x=*r2++                 /* p = e2*d2; x = e1          */
        a0=a0+p p=x*y   y=*r3++                 /* p = e1*d2; y = b1          */
        a0=rnd(a0)                              /* db1=rnd(e1*d1+e2*d2)       */
        a0=a0+y         y=*r3--                 /* b1(1)=b1(0)+db1; y = b2    */
        a1=a1-p         *r3++=a0                /* update b1 in ram           */

        a1=rnd(a1)                              /* db2=rnd(e2*d1-e1*d2)       */
        a1=a1+y         y=*r1++                 /* b2(1)=b2(0)+db2; y = d1    */
        *r3++=a1                                /* update b2 in ram           */
        }
                p=x*y   x=*r2--
        a0=p    p=x*y   y=*r1
        a1=p    p=x*y   x=*r2
        a0=a0+p p=x*y   y=*r3++
        a0=rnd(a0)
        a0=a0+y         y=*r3--
        a1=a1-p         *r3++=a0
        a1=rnd(a1)
        a1=a1+y         *r0--
        *r3++=a1

        y=*r0--                                 /* Update the feedback eqlzr  */
        *r1--=y                                 /* delay line elements d1(0), */
        y=*r0                                   /* d2(0).                     */
        *r1=y

        r0=pnt
        *r0zp:r1                                /* Exchange the pointer       */
        *r0zp:rb                                /* settings for the forward   */
        *r0zp:re                                /* equalizer tap update.      */
```

```
/* forward equalizer coefficient adaptation using the LMS algorithm          */
lms2:   r3=cf1
        x=*r2++                          /* x = e1                           */
        y=*r1++                          /* y = x1                           */
   do 2 {
                p=x*y   x=*r2--          /* p = e1*x1; x = e2                */
        a0=p    p=x*y   y=*r1++          /* p = e2*x1; y = x2                */
        a1=p    p=x*y   x=*r2++          /* p = e2*x2; x = e1                */
        a0=a0+p p=x*y   y=*r3++          /* p = e1*x2; y = c1                */
        a0=rnd(a0)                       /* dc1 = rnd(e1*x1+e2*x2)           */
        a0=a0+y         y=*r3--          /* c1(1) = c1(0)+dc1; y = c2        */
        a1=a1-p         *r3++=a0         /* update c1 in ram                 */
        a1=rnd(a1)                       /* dc2 = rnd(e2*x1-e1*x2)           */
        a1=a1+y         y=*r1++          /* c2(1) = c2(0)+dc2; y = x1        */
        *r3++=a1                         /* update c2 in ram                 */
        }

                p=x*y   x=*r2--
        a0=p    p=x*y   y=*r1--
        a1=p    p=x*y   x=*r2++
        a0=a0+p p=x*y   y=*r3++
        a0=rnd(a0)
        a0=a0+y         y=*r3--
        a1=a1-p         *r3++=a0
        a1=rnd(a1)
        a1=a1+y         *r0++
        *r3++=a1

        y=*r0--                          /* Transfer new data to forward     */
        *r1++=y                          /* equalizer delay line elements    */
        y=*r0++j                         /* x1(0), x2(0).                    */
        *r1--=y
        sdx=*r0--                        /* output imag. data to SIO         */
        goto loop                        /* return to main loop              */

/* subroutine: dec - 2-bit quantizer          */
dec:    r3=trh                           /* r3 -> trh                        */
        r2=qtz                           /* r2 -> qtz                        */
        y=*r3++                          /* fetch threshold level -0.2       */
        a0-y                             /* to y and compare with a0         */
        if le return                     /* if greater, increment r2         */
```

```
        *r2++
        y=*r3++                         /* compare with threshold 0,     */
        a0-y                            /* if greater,increment r2       */
        if le return

        *r2++
        y=*r3                           /* compare with threshold 0.2    */
        a0-y                            /* if greater, increment r2      */
        if le return

        *r2++
        return

CI:     int     0                       /* forward equalizer coef.       */
CQ:     int     0
        int     0
        int     0
        int     1.0
        int     0
BCI:    int     0                       /* feedback equalizer coef.      */
BCQ:    int     0
        int     0
        int     0
        int     0
        int     0
        int     0
        int     0
        int     0
        int     0
        int     0
TRH:    int    -0.2                     /* threshold levels              */
        int     0.0
        int     0.2
QTZ:    int    -0.3                     /* quantization levels           */
        int    -0.1
        int     0.1
        int     0.3
U:      int    -0.05                    /* -(convergence factor)         */
```

For additional information, contact
your AT&T Account Manager, or call:

- AT&T Microelectronics
  Dept. 52AL330240
  555 Union Boulevard
  Allentown, PA 18103
  1-800-372-2447

  In Canada, call:
  1-800-553-2448

- AT&T Microelectronics
  AT&T Deutschland GmbH
  Bahnhofstr. 27A
  D-8043 Unterfoehring
  West Germany
  Tel. 089/950 86-0
  Telefax 089/950 86-111

- AT&T Microelectronics Asia/Pacific
  14 Science Park Drive
  #03-02A/04 The Maxwell
  Singapore 0511
  Tel. (65) 778-8833
  FAX (65) 777-7495
  Telex RS 42898 ATTM

- AT&T Microelectronics
  AT&T Japan Ltd.
  31-11, Yoyogi 1-chome
  Shibuya-ku, Tokyo 151
  Japan
  Tel. (03) 5371-2700
  FAX (03) 5371-3556

- AT&T Microelectronica España
  C/ .Albacete, 5 - 2.ª planta
  28027 Madrid
  Spain
  Tel. 404 60 12
  FAX 404 34 69
  Telex 41494 AMESP

WE is a registered trademark of AT&T.

AT&T reserves the right to make changes to the product(s), software, or circuit(s) described herein without notice. No liability is assumed as a result of their use or application. No rights under any patent accompany the sale of any such product or circuit.

Copyright © 1989 AT&T
All Rights Reserved
Printed in U. S. A.

June 1989

AP89-009DMOS

AT&T
The right choice.