Application Note

AT&T

Noise Generation Routines Using the AT&T *WE*® DSP16 Digital Signal Processor

Contents **Page**

1. Introduction

Digital signal processors (DSPs) have the capability to execute sequences of instructions with precision, accuracy, and repeatability. However, there are times when controlled amounts of variability are required or desirable when performing calculations. Variability may be inserted by using random numbers. When sequences of random numbers are used as signals, they are described as noise sources. Applications for noise sources include:

1. Equalization of channels for modems [1].
2. Dithering of speech and video signals with deterministic noise prior to quantization in order to increase the perceived quality of the receive signal [2].
3. Replacing gaps in speech signals where silence has been removed for transmission or storage efficiencies, allowing more natural sound when reconstructed with background noise at the same level as the original speech [3].
4. Improving performance of certain signal-processing algorithms by adding low-level noise to the signal being analyzed [2].
5. Applications requiring random numbers such as random tree search, random codebook search, and Monte-Carlo analysis.

There are two basic-methods of generating random number sequences. The first is to measure some truly random quantity such as the time between counts of a Geiger Counter, the voltage across a large resistance, or the phase difference between two free-running oscillators. Although these examples of measurement will produce random number sequences, they suffer from several deficiencies:

1. Portability. These methods may not be practical for all situations.
2. Repeatability between systems. It would be difficult to guarantee that two instances of such physically-based random number sources produce sequences with the same characteristics.
3. Repeatability over time. There are times when the same random number sequence is required to evaluate modifications made to an algorithm.

These deficiencies are eliminated if the random sequences are generated by a deterministic processor (the second general method of generating random number sequences). However, a new problem arises, namely insuring that the sequence generated is truly random.

This application note will describe noise generation using the DSP16 Digital Signal Processor. Following this section, section two describes several random number generation algorithms; section three describes methods of processing a random number sequence to produce different probability distributions; and section four describes methods of modifying the spectrum of the noise signal generated from a random number generator. DSP16 code is given and described for all of these techniques.

2. Random Number Generation

The subject of obtaining random numbers by using digital processors began in the late 1940's with John von Neuman's famous "middle of the squares" algorithm [4]. A form of the algorithm with an example is as follows:

1. Start with a two-digit number.
2. Square the number.
3. Return the middle two digits of the assumed, four-digit number, and use this value as the start for the next iteration.

Several sequences are shown below:

$10 \rightarrow 10 \rightarrow 10 \rightarrow 10 \ldots$

$20 \rightarrow 40 \rightarrow 60 \rightarrow 60 \ldots$

$55 \rightarrow 2 \rightarrow 0 \rightarrow 0 \ldots$

$33 \rightarrow 8 \rightarrow 6 \rightarrow 3 \rightarrow 0 \rightarrow 0 \ldots$

As can be seen, these examples eventually degenerate to a sequence of a single number — not very random. Since John von Neuman's algorithm, there has been much research into random number generation algorithms as well as tests to determine if the resulting sequences are truly random. To date, the definitive reference on random numbers is *The Art of Computer Programming* [4].

Linear Congruential Method

The most widely used method for generating random numbers is called the linear congruential method [4]. It is based on the recursive formula:

$$X[n+1] = (A\,X[n] + C) \bmod M$$

(Equation 1)

Where,

$X[n+1]$	is the next random number in the sequence
$X[n]$	is the current number in the random sequence
A	is the multiplier, $0 \leq A < M$
C	is the increment, $0 \leq C < M$
M	is the modulus, $M > 0$

The first value X[0] is called the seed number and is a value chosen between 0 and (M – 1). The subsequent numbers in the sequence are derived by iterating Equation 1.

The parameters M, C, and A must be selected with great care. Properly selected values will generate a sequence with good random properties; poorly selected values will generate poor sequences. There are well-founded theories on the selection of these parameters; generally, the larger the value of M, the better the random sequence. Also, since the maximum period of the sequence is of length M, a larger value of M results in a potentially longer period.

The range of values of X[n] is from 0 to (M – 1), which limits the values of M, A, and C such that (A X[n] + C) does not overflow the processor. In the case of the DSP16, overflow occurs at 2^{31-1}.

Several good sets of values are shown in Table 1. A C-language program that implements Equation 1 is shown in Appendix 1. The first 50 numbers generated using the first set of parameters are shown in Table 2, and numbers from the fourth set are shown in Table 3.

Table 1. Suggested Parameters for Implementing Equation 1 With the DSP16

M	A	C	Reference
134,456	8,121	28,411	[5]
243,000	4,561	51,349	[5]
259,200	7,141	54,773	[5]
1,048,576	2,045	1	[1]

Table 2. First 50 Iterations of Linear Congruential Method With M = 134,456; A = 8,121; and C = 28,411

Iteration	Number X[n]	
[n]	Decimal	Hex
0	75432	126a8
1	30147	75c3
2	7822	1e8e
3	87641	15659
4	85364	14d74
5	14319	37ef
6	8570	217a
7	11629	1b40d
8	65168	fe90
9	38923	980b
10	16038	3ea6
11	119601	1d331
12	132444	2055c
13	92591	169af
14	81970	14032
15	15125	3b15
16	100208	18770
17	89867	15f0b
18	11150	2b8e
19	88673	15a61
20	129964	1fbac
21	120911	1d84f
22	14474	388a
23	57221	df85
24	40216	9d18
25	28923	70fb
26	17462	4436
27	120689	1d771
28	93996	16f2c
29	63215	f6ef
30	44418	ad82
31	1541	605
32	38464	9640
33	53267	d013
34	64766	fcfe
35	1225	4c9
36	26892	690c
37	61799	f167
38	108298	1a70a
39	39773	9b5d
40	61632	f0c0
41	96651	1798b
42	111510	1b396
43	39961	9c19
44	109364	1ab34
45	91575	165b7
46	32850	8052
47	42557	a63d
48	81888	13fe0
49	21483	53eb

Table 3. First 50 Iterations of Linear Congruential Method With M = 1,048,576; A = 2,045; and C = 1

Iteration	Number X[n]	
[n]	Decimal	Hex
0	104242	19732
1	313963	4ca6b
2	325824	4f8c0
3	464321	715c1
4	575166	8c6be
5	760775	b9bc7
6	746668	b64ac
7	209405	331fd
8	414218	6520a
9	874979	d59e3
10	461400	70a58
11	893177	da0f9
12	976150	ee516
13	786623	c00bf
14	128452	1f5c4
15	540341	83eb5
16	846818	cebe2
17	543835	84c5b
18	652016	9f2f0
19	632625	9a731
20	823918	c926e
21	899255	db8b7
22	822748	c8ddc
23	603757	9366d
24	509114	7c4ba
25	950739	e81d3
26	201352	31288
27	723049	b0869
28	143046	22ec6
29	1024943	fa3af
30	953588	e8cf4
31	784677	bf925
32	343186	53c92
33	318027	4da4b
34	248096	3c920
35	894113	da4a1
36	793118	c1a1e
37	827815	ca1a7
38	480012	7530c
39	157405	266dd
40	1028970	fb36a
41	800195	c35c3
42	620216	976b8
43	613337	95bd9
44	177270	2b476
45	758431	b929f
46	147492	24024
47	679829	a5f95
48	887106	d8942
49	95291	1743b

Example 1: Linear Congruential Method Using the DSP16

Program Segment 1 shows a DSP16 program that implements the random number generation algorithm given in Appendix 1. Program Segment 1 is described below.

The parameters M, A, and C are stored in ROM, and variables XN and XNP1 are stored in RAM. Since the DSP16's RAM and ROM are 16-bits wide, several of the values are stored in two locations. As mentioned, M should be as large as possible; therefore it is stored in two ROM locations (MH and ML). Since the generated random numbers range from 0 to (M – 1), X[n] is also stored in two RAM locations (XNH and XNL).

The first operation is to multiply X[n] by A. Since the DSP16's multiplier uses 16-bit operands, the multiplication is performed in two stages in order to implement extended precision arithmetic. A is first multiplied by the lower half of XN (XNL). Because the most significant bit of XNL can be a 1, XNL can be interpreted as a negative number and the multiplication would produce an incorrectly signed result. To alleviate this possibility, XNL is tested. If XNL is interpreted as negative, the value A is shifted up sixteen bits and added before the multiplication. The product (A XN) is stored in accumulator a0. A is then multiplied by XNH. This partial product is shifted up sixteen bits and then added to the lower partial product. The increment C is then added to the complete product, resulting in (A XN + C).

The next step is to take the modulus of this quantity. The DSP16 does not have an intrinsic quotient/remainder function, so the program loop "do 14" is used to perform the modulus operation. The modulus (in this case 134,456) is shifted to the left as much as possible without producing an overflow (in this case to 1,101,463,552). This value is then tested against the argument (A Xn + C). If the shifted modulus is less than (A Xn + C), it is subtracted from (A Xn + C), otherwise the process continues to the following step. The modulus is shifted once to the right and then tested again. This step is iterated once more than the number of shifts used to produce the shifted modulus. After these iterations, the remainder is the modulus of the argument. In order to perform the 31-bit subtraction, the shifted modulus must be written to RAM and read back into the y register.

Finally, the newly generated random number must be normalized. In typical applications, the numbers should range from 0 to 2^{15}, or -2^{15} to 2^{15}. Most of the moduli in Table 1 are not powers of 2, and hence converting these numbers to a useful range requires scaling the number. This scale factor is the last item in the ROM data-base. The resulting number returned to the calling program ranges from -2^{15+1} to 2^{15-1} and is held in location "udev."

The last set of numbers in Table 1 has a modulus that is a power of 2. This greatly simplifies the generation of the random numbers, as the modulus operation involves simply masking out the most significant bits. In addition, scaling the number to the range $\pm 2^{15}$ requires a simple shift. The program segment that works with the last set of numbers in Table 1 is given in Program Segment 2.

Although there are benefits to using a modulus that is a power of 2, the "n" least significant bits of the numbers generated in this manner repeat with a period of 2^n. This is seen by examining the hexadecimal representation of the random numbers in Table 3. Note that this is not the case with the data in Table 2.

Program Segment 1. Linear Congruential Method

```
/*      The following is the program segment for the first DSP example.  This program     */
/*      segment implements the linear congruential method of random number generation, i.e.  */
/*                                                                                         */
/*                        x(n+1) = (a*x(n) + c) mod m                                      */
/*                                                                                         */
/*      Inputs:RAM variables xnl, xnh - seed value                                         */
/*             ROM locations a,c,m - parameters for algorithm.                             */
/*      Outputs:RAM variables xnl, xnh - next random number                                */
/*             RAM variable udev - normalized 16 bits random number                        */
/*             a0 - random number                                                          */
/*             a1 - 16 bit normalized random number                                        */

        /*declarations of ram variables*/
.ram
xnl:    int     /*low 16 bits of seed number/previously generated number */
                /*xn gets overwritten with xnp1 for use at the next call */
xnh:    int     /*high 16 bits of number                                 */
tmpl:   int
tmph:   int
udev:   int     /*returned uniform deviate in ram                        */
.endram
udev0:
                pt=lcds                                                 /*set rom pointer     */
                r1=xnl                                                  /*point to old number */
                a0=y                                                    /*just to clear a0    */
                a0=a0-y                     y=*r1++     x=*pt++         /*y=xnl, x=a          */
                a1=y                                                    /*a1=xnl              */
                            p=x*y           y=*r1                       /*p=lo protd, y=xnh   */
        if pl   goto cont0                                              /*if xnl pl, continue */
                a0=x                                                    /*load x to a0        */
cont0:
                a0=a0+p     p=x*y                                       /*a=unsign mul,p=hi prod*/
test0:          a1=p                                                    /*a1=hi prod, a0=lo prod*/
                a1=a1<<16                                               /*a1=hi prod in MSW    */
                *r1=a1
                                            y=*r1--                     /*to get to upper bits */
                a0=a0+y                                                 /*now have a*xn       */
                                            y=*r1       x=*pt++         /*y=<>, x=c           */
                                            y=0x1
                            p=x*y
                a0=a0+p                                                 /*a0 = a*x+c          */

        /*a0 now has sum right justified    */
                r1=tmpl
                                            y=*r1       x=*pt++         /*y=<>,x=ml           */
                                            *r1++=x                     /*tmpl=ml             */
                                            y=*r1       x=*pt++
                                            *r1=x
                                            y=*r1--                     /*transfer to y       */
                                            yl=*r1

/*      a0 has the sum from which the modulus will be taken. y has the modulus shifted to     */
/*      the maximum magnitude.  The number of shifts depends upon the modulus.  The modulus   */
/*      134,456 shifted 13 times results in 0x41a70000 = 1,101,463,552.  This is the maximum  */
/*      shifted value less than 2^31-1.  The do loop will calculate the modulus as follows:   */
/*             if a0 > y, subtract y                                                          */
/*             shift y right (>>) by one                                                      */
/*             repeat until y is the true modulus value                                       */
/*      The number of iterations is one plus the number of shifts in obtaining the maximum    */
/*      modulus.                                                                              */
```

```
        do 14  {
               a1=a0-y                                                      /*test for magnitude */
               if pl a0=a1                                                  /*subtract           */
cont1:         a1=y
               a1=a1>>1                                                     /*shift down         */
               *r1++=a1                                                     /*store high half    */
               a1=a1<<16                                                    /*get lsbs           */
               *r1--=a1                                                     /*store low half     */
               y=*r1++                                                      /*get high half in y */
               y1=*r1--                                                     /*get low half       */
               }

        /*a0 now has modularized value, store it into xnp1h, xnp1l          */
               a1=a0                                                        /*for use later      */
               r1=xnh
               *r1--=a0
               a0=a0<<16
               *r1=a0

        /*now scale the number for +/- 2^15; scale factor (1.949)*/
        /*depends on ratio of value of modulus to next power of 2*/
scl1:          a1=a1<<8                                                     /*get to MSW         */
               a1=a1<<4
                                            y=*r1          x=*pt++          /*x=scale factor     */
                                            y=a1                            /*y=number           */
                              p=x*y                                         /*scale to 16 bits   */
               a0=p
               a0=a0<<4
               r1=udev
               *r1=a0                                                       /*store in XRET      */
udev0e:        return

lcds:
        int    8121           /* A:multiplier                  */
        int    28411          /* C:increment                   */
        int    0x0000         /* ML:low half of shifted modulus */
        int    0x41a7         /* MH:high half of shifted modulus */
        int    1.949          /* scale                         */
```

Program Segment 2. Linear Congruential Method

```
/*      The following is the program segment for the second DSP16 example.  This program      */
/*      segment implements the linear congruential method of random number generation, i.e.   */
/*                          x(n+1) = (a*x(n) + c) mod m                                        */
/*      This program is optimized for modulus of power of two.  Assumed modulus is 2^20.       */
/*                                                                                             */
/*      Inputs:RAM variables XNL, XNH - seed value                                             */
/*             ROM locations a,c - parameters for linear congr. algo.                          */
/*      Outputs:RAM variables XNL, XNH - next 20 bit random number                             */
/*             RAM variable udev - normalized 16 bits random number                            */
/*             a0 - 20 bit random number                                                       */
/*             a1 - 16 bit normalized random number                                            */
```

```
        /*declarations of ram variables     */
.ram
xnl:    int     /*low 16 bits of seed number/previously generated number */
                /*xn gets overwritten with xnp1 for use at the next call */
xnh:    int     /*high 16 bits of number                                 */
udev:   int     /*returned uniform deviate in ram                        */
.endram

udev1:
                pt=lcds                                             /*set rom pointer    */
                r1=xnl                                              /*point to old number*/
                a0=y                                                /*just to clear a0   */
                a0=a0-y                  y=*r1++      x=*pt++       /*y=xnl, x=a         */
                a1=y                                                /*a1=xnl             */
                            p=x*y        y=*r1                      /*p=lo protd, y=xnh  */
        if pl   goto cont                                           /*if xnl pl, continue*/
                a0=x                                                /*load x to a0       */
cont:
                a0=a0+p     p=x*y                                   /*a=unsign mul,p=hi prod*/
test1:          a1=p                                                /*a1=hi prod, a0 lo prod*/
                a1=a1<<16                                           /*a1=hi prod in MSW  */
                *r1=a1
                y=*r1--                                             /*to get to upper bits */
                a0=a0+y                                             /*now have a*xn      */
                                         y=*r1        x=*pt++       /*y=<>, x=c          */
                                         y=0x1
                            p=x*y
                a0=a0+p                                             /*a0 = a*x+c         */
        /*a0 now has sum right justified     */
                y=0xf                                               /*gen. 20 bit mask in y */
                yl=0xffff                                           /*LSW of mask        */
                a0=a0&y                                             /*mask = modulus     */

        /*a0 now has modular value, store it into xnp1h, xnp1l    */
                a1=a0                                               /*save for later     */
                *r1++=a1l                                           /*r1=xnl             */
                *r1++=a1                                            /*r1=xnh             */
                                                                    /*now r1=udev        */

        /*move to sign plus 15-bits of precision, and store       */
        /*we assume that the multiplier is 2^20                   */
                a0=a0<<8                                            /*2^28               */
                a0=a0<<4                                            /*2^32               */
                *r1=a0                                              /*store in RAM       */
udev1e:         return

lcds:
        int     2045        /* A:multiplier                       */
        int     1           /* C:increment                        */
        int     0           /* M:modulus is hardwired in code     */
```

Example 2: Built-In Random Number Generator For the DSP16

The DSP16 has an internal random bit generator that can be used in another method of generating random numbers. The random bit generator is based on a ten-stage, pseudo-random bit sequence generator and can be reset by writing to the pi register. The output of the generator is observed by performing a conditional test using the heads-or-tails condition. These tests indicate if the output of the internal generator is a 1 or a 0. In addition, testing the output generates a new output from the bit generator.

This method is much simpler than using the linear congruential algorithm; however, caution must be used when concatenating random bits to develop random numbers, as the theory in determining the quality of the resulting random numbers has not been fully developed. In addition, the linear congruential method can be restarted in a known state, restarted in a different state, or can use a different set of parameters to generate a different sequence.

Program Segment 3 shows how to generate random numbers by using the internal DSP16 random bit generator. In this method, Accumulator 0 is first cleared. Then, bit 1 of the upper half of Accumulator 0 is randomly set, depending on the state of the internal random bit generator. Accumulator 0 is then shifted left one location. This process repeats 15 more times. Finally, the 16-bit number is stored in RAM for return back to the calling routine.

Program Segment 3. Built-In Generation Method

```
/*      The following is the program segment for the third DSP example.  This program          */
/*      segment uses the built-in random bit generator to generate random numbers.              */

        /*declarations of ram variables     */
.ram
udev:   int     /*returned random number     */
.endram

unif0:
                a0=y                        /*clear a0            */
                a0=a0-y
        do 16   {
                if heads a0h = a0h + 1      /*randomly set LSB    */
                a0 = a0 << 1                /*shift               */
                }
                r1=udev
                *r1=a0
endran:         return
```

3. Noise Generation With Specific Probability Density Function

One of the most important characteristics used in describing a random number sequence is the probability density function (PDF). The PDF describes the probability that a sample of a given amplitude or range of amplitudes will be generated. Examples of different PDFs are uniform, exponential, Poisson, and Gaussian [6].

The DSP random number generators in Examples 1 and 2 will produce numbers with a uniform PDF. A uniform PDF is shown in Figure 1 and is described as:

When $\frac{-a}{2} \leq x \leq \frac{a}{2}$

$$\mathrm{PDF}(x) = \frac{1}{a}$$

Otherwise,

$$\mathrm{PDF}(x) = 0$$

(Equation 2)

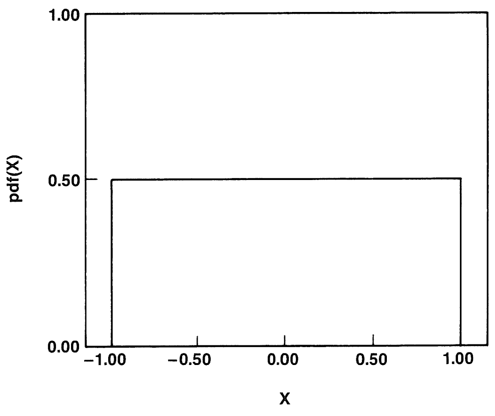

Figure 1. Uniform Probability Density Function

A Gaussian PDF of unit variance and 0 mean is shown in Figure 2 and is described as:

When $-\infty < x < +\infty$

$$\mathrm{PDF}(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$$

(Equation 3)

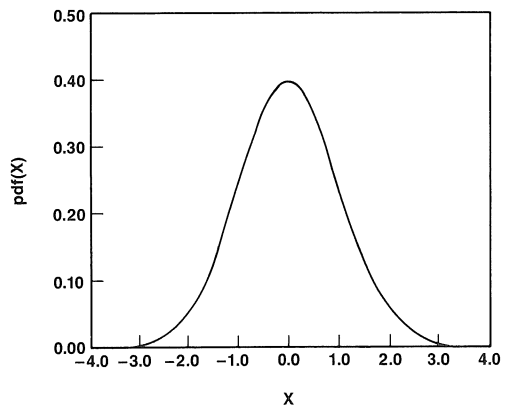

Figure 2. Gaussian Probability Density Function

To determine the probability of a range of numbers, the PDF is integrated over that range of numbers:

$$P(a < x < b) = \int_{y=a}^{y=b} \mathrm{pdf}(y)dy$$

(Equation 4)

Associated with the PDF is the cumulative distribution function (CDF). The CDF is the integral of the PDF:

$$\mathrm{CDF}(x) = \int_{y=-\infty}^{y=x} \mathrm{pdf}(y)dy$$

(Equation 5)

The CDFs for the uniform and Gaussian PDFs are shown in Figures 3 and 4. The CDF evaluated at X indicates the probability that the random number will be less than or equal to X. The CDF can be used to determine the probability of a number occurring within a range by evaluating the CDF at the extremes of the range and by then taking the difference:

$$P(a < x < b) = \mathrm{CDF}(b) - \mathrm{CDF}(a)$$

(Equation 6)

Equations 5 and 6 are true for continuous distributions. For discrete distributions, the integration is replaced by summation.

Due to the finite representation of numbers in processors, there is a bound on all types of distributions, even those that theoretically have no upper or lower bound. However, if the range of the random numbers is appropriately limited, this boundary effect can be minimized.

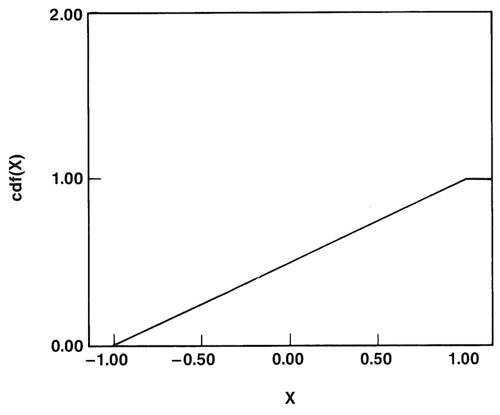

Figure 3. Uniform Cumulative Distribution Function

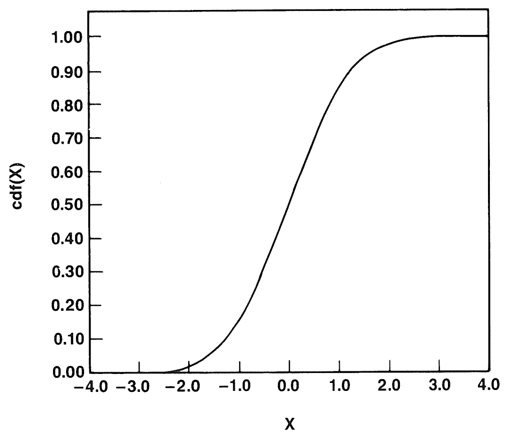

Figure 4. Gaussian Cumulative Distribution Function

Generation of Random Numbers With Different Probability Functions

As mentioned, the DSP random number generators in Examples 1 and 2 will generate sequences with uniform PDFs. Uniformly distributed numbers (also called uniform deviates) are used in simulations of games of chance (dealing cards, rolling dice, etc.) and random searches of tables. If these numbers were used to construct an analog waveform, it would be called white- or broadband-noise. White-noise is used in simulating linear quantization noise or determining the frequency response of a system.

There are times when specific PDFs are desired for white-noise. The most commonly encountered is a Guassian PDF, which is used to model component variations or some types of noise sources. Many random sequences are modeled with Gaussian PDFs since it is so well known and understood.

A significant operation in random number generation is transforming the uniform random numbers to ones with a specific PDF. References [4] and [6] describe a procedure for this operation with more detail.

Assume a uniformly distributed deviate X. A variable Y is needed with a PDF of PDFy and a CDF of CDFy. It turns out that the transformation from X to Y is:

$$Y = CDF^{-1}(X) = G(X)$$

(Equation 7)

where G( ) is the inverse of the desired cumulative distribution function.

Example 3: Exponential Distribution

The exponential distribution is described by:

When $X \geq 0$

$$PDF(X) = A\, e^{\frac{-X}{A}}$$

(Equation 8)

When $X \geq 0$

$$CDF(X) = 1 - e^{\frac{-X}{A}}$$

(Equation 9)

with parameter A determining the "spread" of the distribution. The function G(X) used to generate exponential deviates from uniform deviates can easily be derived from Equation 9:

When $X \neq 0$

$$X = CDF(Y)$$

$$X = 1 - e^{\frac{-Y}{A}}$$

$$e^{\frac{-Y}{A}} = 1 - X$$

$$\frac{-Y}{A} = \ln(1 - X)$$

$$Y = -A \ln(1 - X)$$

$$Y = G(X) = -A \ln(X)$$

(Equation 10)

If X is uniformly distributed from 0 to 1, the function $G(X) = -A \ln(1 - X)$ will produce a sequence that is exponentially distributed. A simplification can be made by observing that if X is uniformly distributed (see Figure 1), so is $(1 - X)$, resulting in the last form of the equation. The value $X = 0$ should be avoided in this case, since this would result in an undefined value of y.

Example 4: Gaussian Distribution

Gaussian deviates have a PDF as described in Equation 3. The following describes deviates of 0 mean and standard deviation of 1. The CDF is:

$$\mathrm{CDF}(X) = \frac{1}{\sqrt{2\pi}} \int_{t=-\infty}^{t=x} e^{(\frac{-t^2}{2})}\, dt$$

(Equation 11)

The integral is similar to the error function [6], and can be re-written as:

When $x \geq 0$

$$\mathrm{CDF}(X) = \frac{1}{2} + \mathrm{erf}(X)$$

When $x < 0$

$$\mathrm{CDF}(X) = -\mathrm{erf}(X)$$

(Equation 12)

erf(X) does not have a closed-form solution but is well tabulated [6, 7]. The inverse of CDF( ) is:

When $x \geq 0$

$$Y = \mathrm{CDF}^{-1}(X) = \mathrm{erf}^{-1}(X - \frac{1}{2})$$

When $x < 0$

$$\mathrm{erf}^{-1}(-X)$$

(Equation 13)

Since the function does not have a closed-form solution, neither does the inverse. However, the inverse can be obtained from look-up tables. The inverse of the erf function is shown in Figure 5.

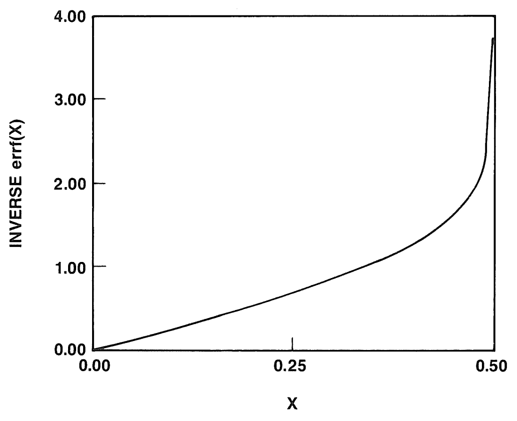

Figure 5. Inverse Error Function

Generation of Random Numbers With Gaussian PDF

As seen above, the generation of Gaussian deviates from uniform deviates is not trivial. Several algorithmic approaches are described in Reference [1]. The simplest is called the polar method. Since it requires a division, square root, and logarithm function, it is not well suited for the fixed-point DSP16. The simplest and most straight-forward method is to pre-compute a table of the inverse CDF and look-up the Gaussian deviates with the random deviates as indices into the table.

Table 4. Table Used in Program Segment 4

integer	x-value	y-value
348	0.015622	0.039368
1040	0.046866	0.117493
1743	0.078109	0.196838
2456	0.109353	0.277405
3186	0.140597	0.359802
3943	0.171841	0.445251
4721	0.203084	0.533142
5543	0.234328	0.625916
6413	0.265572	0.724182
7353	0.296816	0.830383
8380	0.328059	0.946350
9537	0.359303	1.076965
10888	0.390547	1.229553
12548	0.421791	1.416931
14834	0.453034	1.675110
19050	0.484278	2.151184

The generation of the Table 4 is done by the C-language program in Appendix 2. The range from 0 to 0.5 is divided into NINTS segments, and the midpoint of each segment is passed on as the argument to C function erfinv( ), which evaluates the inverse error function.

Function erfinv( ) calculates the inverse error function by performing a binary search on the error function, erf( ). The domain of the error function is limited from 0 to 3.7, which includes 99.99% of all the numbers occurring in a true Gaussian random number sequence. The mid-point of this section is evaluated, and compared to the desired value X passed as an argument. If larger than X, the low end of the domain is moved to the mid-point; otherwise the high-end is moved to the mid-point. This iteration continues until the high- and low-ends are separated by 0.001. At this point, the previous mid-point is taken as the solution and is returned. In addition, the actual x-value that corresponds to the returned y-value is also made available.

Function erf( ) is calculated by performing trapezoidal integration over the Gaussian PDF function. The limits of the integral are divided into 100 sections. The PDF is then evaluated at the mid-point of each section, weighted, and summed to produce the final result.

The C-language routines for erf( ) and erfinv( ) are in Appendix 4.

Example 5: Generation of Noise With Gaussian PDF and the DSP16

Program Segment 4 shows the DSP16 subroutine for calculating a random number sequence with Gaussian distribution. The distribution will have 0 mean and a standard deviation of 1. The method used is the table look-up described in the previous section. A table with 16 entries is used.

Function GDEV0 calls a random number generator, which can be that of either Program Segment 1 or 2. The most significant bit of the returned uniform deviate, UDEV, determines the sign of the Gaussian deviate, GDEV. The next four bits are used as an index into table U2G, and the Gaussian deviate is returned. The sign is then appended, and the signed Gaussian deviate is returned to the main program.

Program Segment 4. Look-Up Table Method

```
/*      This program segment uses table look-up to generate Gaussian distributed deviates.    */

/*ram declarations    */
.ram
oldpr: int    /*old program return value   */
sign:  int    /*sign of deviate            */
gret:  int    /*Gaussian deviate           */
.endram

gdev0:
       /*save program return register for calling routine       */
                                                     r1=oldpr
                              *r1++=pr                                        /*save pr            */

                      /*get uniform deviate */
       call unif0     /*calculate udev from program segment 3    */

       /*manipulate returned value for index      */
              r1=sign
              r3=udev
              pt=gdevt

              y=1                                                             /*pre-load if pos    */
              a1=y

              y=*r3                                                           /*get unif           */
              a0=y
              y=0xffff                                                        /*set y -            */
       if mi  a1=y                                                            /*if mi, load fs     */
              *r1=a1

              /*get the next four bits    */
              y=0xf
              a0=a0<<1
              a0=a0>>4
              a0=a0>>8                                                        /*next 4 bits        */
              a0=a0&y
              i=a0                                                            /*load index         */
                                            y=*r1          x=*pt++i           /*incr pt            */
                                            y=*r1++        x=*pt++            /*get real           */
                              p=x*y                                           /*y has sign         */
              a0=p
              a0=a0<<16
              *r1--=a0                                                        /*return             */
              /*restore pr   */
              *r1--=a0                                                        /*just for --        */
              pr=*r1
              return
# include "pseg3.s"
gdevt:
                      /* x value    y value       */
       int    348     /* 0.015622   0.039368      */
       int    1040    /* 0.046866   0.117493      */
       int    1743    /* 0.078109   0.196838      */
       int    2456    /* 0.109353   0.277405      */
       int    3186    /* 0.140597   0.359802      */
       int    3943    /* 0.171841   0.445251      */
int     4721   /* 0.203084    0.533142        */
int     5543   /* 0.234328    0.625916        */
int     6413   /* 0.265572    0.724182        */
int     7353   /* 0.296816    0.830383        */
int     8380   /* 0.328059    0.946350        */
int     9537   /* 0.359303    1.076965        */
int     10888  /* 0.390547    1.229553        */
int     12548  /* 0.421791    1.416931        */
int     14834  /* 0.453034    1.675110        */
int     19050  /* 0.484278    2.151184        */
```

Improved Generation of Random Numbers With Gaussian PDF

One limitation of the above example is that only 32 different Gaussian deviates would ever be produced. This would be useful in only very limited applications. One solution would be to increase the table size. A second solution is as follows: In the previous example, each segment of the inverse error function was approximated by a constant value. An improvement would be to model the entire inverse function by a piece-wise linear approximation.

Example 6: Improved Generation of Noise With Gaussian PDF Distribution and the DSP16

To determine a Gaussian deviate one would first pick a random segment, as in the previous example, and then pick a random point within the range of that segment.

Table 5 is generated by the program in Appendix 3, with the table entries being the starting points of each segment. There is an additional entry for the ending point of the last segment. In reality, this last point should be positive infinity, but due to finite arithmetic on the DSP16, the maximum positive value (0x7 f f f) is used. The numbers are scaled in the table such that 99.99% of the theoretical range is covered.

Table 5. Table Used in Program Segment 5

integer	x-value	y-value
0	0.000000	0.000000
694	0.031244	0.078430
1391	0.062488	0.157166
2099	0.093731	0.237122
2818	0.124975	0.318298
3559	0.156219	0.401917
4326	0.187463	0.488586
5126	0.218706	0.578918
5970	0.249950	0.674133
6872	0.281194	0.776062
7856	0.312438	0.887146
8942	0.343681	1.009827
10186	0.374925	1.150208
11667	0.406169	1.317444
13580	0.437413	1.533508
16483	0.468656	1.861267
32767	0.499900	3.700000

Program Segment 5 calls a function that generates a uniform deviate, U. The sign of UDEV is stored for later use. The next four bits are used to pick a segment. The next eight bits are extracted and stored. These will be used to pick a random point on the straight line approximation. An alternative would be to pick a second uniform deviate and use its eight most significant bits for the random point. This would be particularly appropriate if there is concern about the randomness of the least significant bits of the uniform deviates.

Next the difference between the starting value of the chosen segment and the next segment is calculated and multiplied by the stored eight bits. This product is added to the starting value of the segment, and the sign is appended. The signed Gaussian deviate is then returned back to the calling routine.

Program Segment 5. Look-Up Table Method

```
/*      This program segment uses table look-up to generate Gaussian distributed deviates.    */
/*      The value returned is a random interpolation between two table entries.               */

/*ram declarations*/
.ram
oldpr:  int     /*old program return value                          */
sign:   int     /*sign of deviate                                   */
gret:   int     /*Gaussian deviate, and lo value from table         */
diff:   int     /*difference tween lo and high                      */
.endram

gdev0:
        /*save program return register for calling routine          */
              r1=oldpr
              *r1++=pr                                                    /*save pr     */

                    /*get uniform deviate */
        call unif0  /*calculate udev from program segment 3         */

        /*manipulate returned value for index        */
              r1=sign
              r3=udev
              pt=gdevt

              y=1                                                         /*pre-load +1        */
              a1=y

              y=*r3                                                       /*get unif           */
              a0=y
              y=0xffff                                                    /*set y -            */
        if mi a1=y                                                        /*if mi, load fs     */
              *r1++=a1                                                    /*store sign         */

              /*get next few bits   */
              y=0xf
              a0=a0<<1
              a0=a0>>4
              a0=a0>>8                                                    /*next 4 bits        */
              a0=a0&y
              i=a0                                                        /*load index         */

              /*get the lo-end and hi-end of the segment, and get diff */
                                      y=*r1           x=*pt++i            /*incr pt            */
                                      y=*r1           x=*pt++             /*get real           */
                                      y=0x1
                        p=x*y                                             /*p=lo val           */
              a0=p
                                      y=*r1           x=*pt++             /*x=hi val           */
                                      y=0x1
                        p=x*y                                             /*p=hi val           */
              a1=a0-p                                                     /*a1=-diff           */
              *r1++=a0l                                                   /*hold lo            */
              *r1=a1l                                                     /*hold diff          */

        /*get second deviate */
        call unif0  /*calculate udev      */
              r1=diff
        a0=a0>>8                                                    /*8 bits            */
        y=0xff                                                      /*mask              */
        a0=a0&y
        x=a0                                                        /*x=8-bit ran num   */

    /*scale diff by 2nd deviate, add to lo     */
                              y=*r1--                               /*y=-diff           */
                    p=x*y     y=*r1--                               /*p=scl_y y=lo      */
        a0=p
        a0=a0<<8                                                    /*scl to diff       */
        a0=a0-y                                                     /*a0=-lo-scl*diff   */
        a1=-a0                                                      /*change sign       */
                                            x=a1
                              y=*r1++                               /*y=sign            */
                    p=x*y                                           /*add sign          */
        a0=p
        a0=a0<<16                                                   /*scale up          */
        *r1--=a0
        /*restore pr   */
        y=*r1--                                                     /*just to dec r1    */
        pr=*r1
    return
# include "pseg3.s"
gdevt:
                /* x value    y value       */
    int   0     /* 0.000000   0.000000      */
    int   694   /* 0.031244   0.078430      */
    int   1391  /* 0.062488   0.157166      */
    int   2099  /* 0.093731   0.237122      */
    int   2818  /* 0.124975   0.318298      */
    int   3559  /* 0.156219   0.401917      */
    int   4326  /* 0.187463   0.488586      */
    int   5126  /* 0.218706   0.578918      */
    int   5970  /* 0.249950   0.674133      */
    int   6872  /* 0.281194   0.776062      */
    int   7856  /* 0.312438   0.887146      */
    int   8942  /* 0.343681   1.009827      */
    int   10186 /* 0.374925   1.150208      */
    int   11667 /* 0.406169   1.317444      */
    int   13580 /* 0.437413   1.533508      */
    int   16483 /* 0.468656   1.861267      */
    int   32767 /* 0.499900   3.700000      */
```

4. Noise With Specific Spectral Density Characteristics

When a random number sequence is used to generate noise for signal processing, additional characteristics can be described. If the sequence is interpreted as a signal, then the signal has a sampling rate, and hence a spectrum. The spectrum describes the distribution of the energy of the signal over the range of frequencies from 0 to 0.5 times the sampling rate. The spectrum is a second important characteristic of a random number sequence, especially when used to generate noise. For uniform deviates, the spectrum is flat; i.e. the energy is evenly divided across the entire span of frequencies. In certain situations, especially in speech or video applications, noise with a different spectrum is often used. For example, narrow-band noise is often used in testing. Also, pink-noise, which is noise that has a constantly decreasing energy spectrum, might be used to simulate to the spectrum found in falling rain or a waterfall.

Methods of Generation of Different Spectral Functions

The simplest way to generate noise with a specific spectrum is by filtering. Filter generation techniques are quite numerous [8], and techniques have been automated with computer aided design. However, a brief review follows.

There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR) filters. Their names stem from the impulse response duration of the filters.

FIR filters calculate their output based on the current and previous inputs:

$$y(n) = \sum_{k=0}^{k=m} a(k)\, x(n-k)$$

(Equation 13)

IIR filters calculate their output based on the current and previous inputs as well as on the previous outputs:

$$y(n) = \sum_{k=0}^{k=m} a(k)\, x(n-k) + \sum_{j=1}^{j=n} b(k)\, y(n-j)$$

(Equation 14)

Although either FIR or IIR filters can be used, IIR filters are preferred since fewer terms are needed to produce the desired spectrum.

Specific Example in Generation of Spectrally Colored Noise on DSP16

Program Segment 6 shows a DSP16 program that implements a simple first-order, low-pass IIR filter. The coefficient alpha is chosen so that the spectrum of the filtered signal will be flat up to $\frac{1}{10}$ the sampling rate and then decrease with 6 dB per octave.

Program Segment 6. IIR Filter Implementation

```
/*      This program segment implements an IIR filter to filter a random number sequence.    */
/*      The spectrum of the filtered sequence will be low-pass filtered, with a cutoff       */
/*      frequency of 0.1 times the sampling rate.  This program segment assumes that         */
/*      the SAT field in the AUC register is enabled.                                        */

/*ram declarations    */
.ram
oldpr: int      /*old program return value          */
fret:int        /*old filtered signal and new one   */
.endram

fdev:
        /*save program return register for calling routine        */
        r1=oldpr
        *r1++=pr                                            /*save pr           */
        /*get uniform deviate from program segment 3      */
        call unif0

        /*get old value, scale down, filter and add on new deviate     */
                pt=alpha
                r2=udev                                     /*for unif0         */
                r1=fret
                                y=*r1           x=*pt++
                        p=x*y                               /*multiply          */
                a1=p            y=*r2                       /*y=new rv          */

                a0=a0>>1                                    /*a0 from unif0     */
                a0=a0>>1
                                y=a0
                a1=a1+y                                     /*now filtered      */
                a1=a1<<1                                    /*shift to nom.     */
                a1=a1<<1
                *r1=a1                                      /*final answer      */

                /*restore pr  */
                y=*r1--                                     /*just to dec r1    */
                pr=*r1
        return
alpha: int      1.414           /*alpha = .707, determines corner    */
                                /*set to 1.414 for scaling           */
# include "pseg3.s"
```

5. Conclusion

The generation of random numbers and noise is often used in speech and signal processing. This application note has shown that the DSP16 can be used to generate random numbers and noise efficiently for real-time applications.

6. Summary of DSP16 programs

Table 6 shows a summary of the DSP16 program segments described in this application note.

Table 6. Summary of DSP16 Programs

Program Segment	Description	RAM	ROM (ROM + tables)	Cycles (not incl. called routines)	Page
1	Linear Congruential #1	5	54	65	5
2	Linear Congruential #2	3	31	37	6
3	Built-In Random Bit Generation	1	7	38	8
4	Gaussian Deviates #1	3	45 (29 + 16)	39	13
5	Gaussian Deviates #2	4	69 (52 + 17)	70	16
6	IIR filter	2	20	25	19

7. References

[1] Widrow, B., and Stearns, S., *Adaptive Signal Processing*, (Englewood Cliffs, New Jersey: Prentice Hall, 1985).

[2] Jaynat, N. S., and Noll, P., *Digital Coding of Waveforms*, (Englewood Cliffs, New Jersey: Prentice Hall, 1984).

[3] Feher, K., *Advanced Digital Communications*, (Engelwood Cliffs, New Jersey: Prentice Hall, 1987).

[4] Knuth, D. E., *The Art of Computer Programming*, (Reading, Massachusetts: Addison Wesley, 1981).

[5] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes*, (Cambridge: Cambridge University Press, 1986).

[6] Papoulis, A., *Probability, Random Variables and Stochastic Processes*, (New York,: McGraw Hill, 1965).

[7] Hillier, F. S., and Lieberman, G. J., *Introduction to Operations Research*, (San Francisco: Holden-Day, 1968).

[8] Rabiner, L. R., and Gold, B., *Theory and Applications of Digital Signal Processing*, (Englewood Cliffs, New Jersey: Prentice Hall, 1975).

8. Appendices

Appendix 1. Linear Congruential Method

```
/*   LINCON   Linear Congruential Method of Random Number Generation.  This program        */
/*      generates random numbers based on the Linear Congruential Algorithm,               */
/*                                                                                         */
/*                         X(n+1) = (a*X(n) + c) mod m                                     */
/*                                                                                         */
/*     The parameters m, a and c are passed as command line arguments used to generate     */
/*     Table 2:  app1 134456 8121 28411 2 and Table 3:  app1 1048576 2045 1 3              */
main(argc,argv)
int argc;
char *argv[];
{
        long xnp1;      /*x(n+1)       */
        long xn;               /*x(n) */
        long m,a,c;
        int i, tnum;
        float maxval, thisval;

        if (argc < 4)
        {
                printf("Need four arguments, m(l), a(l), c(l), tnum0);
                exit();
        }
/*      printf("%s %s %s0,argv[1], argv[2], argv[3]);      */
        sscanf(argv[1]," %ld",&m);
        sscanf(argv[2]," %ld",&a);
        sscanf(argv[3]," %ld",&c);
        sscanf(argv[4]," %d",&tnum);
/*      printf("m = %ld, a = %ld, c = %ld tnum=%d0,m,a,c,tnum);  */

        /*test for overflow   */
        maxval = 1024.*1024.*1024.*2-1;
        thisval = (m-1.)*a + c;
/*      printf("overflow at %f, maximum value is %f0,maxval,thisval);    */
        if(maxval < thisval)
                printf("Error, this set will overflow0);

        xn = 12357;    /*seed number */
        printf("0                Iteration                                   Number0);
        printf("                                                             DecimalHex0);
        for(i=0;i<50;i++)
        {
                xnp1 = (a * xn + c) % m;
                printf("                 %6d                                 %ld%lx0,i,xnp1,xnp1);
                xn = xnp1;
        }
        printf("0);
        printf("                                                      Table %ld0,tnum);
        printf("                         First 50 Numbers Of Linear Congruential Method0);
        printf("                                      With M=%ld, A=%ld, C=%ld0,m,a,c);
}
```

Appendix 2. Table Generation for Program Segment 4

```
/*  MKTBL4   This program repeatedly calls function errfinv() to generate the contents     */
/*     of a Table 4.  The output is useable as DSP16 assembly code.                         */
main(argc,argv)
int argc;
char *argv[];
{
        double errfinv();
        int nints;     /*number of intervals */
        int i;
        float xval, xval0,xincr;
        float xmax,ymax;
        float yval;
        float tmp;
        int ival,iscl;
        if(argc == 1)
        {
                printf("needs an integer argument0);
                exit();
        }

        sscanf(argv[1]," %d",&nints);       /*get number of intervals    */
/*      printf(" nints = %d0,nints);        */

        xmax = .4999;        /*not to .5; need to limit yvals to 3.7     */
        ymax = 3.7;
        iscl = 1024*32 - 1;  /*scale to 2^15-1                            */

        xincr = xmax/nints;  /*set increment                              */
        xval0 = xincr/2;     /*set starting value                         */
        printf("0);
        printf("                gdevt:0);

        printf("                /*This table is for use with Program Segment 4*/0);
        printf("                                        /*x-value          y-value*/0);
        for(i=0;i<nints; i++)
        {
                xval = xval0 + i*xincr; /*mid point of interval i */
                tmp = errfinv(xval,&yval);
                ival = (yval/ymax)*iscl;
                printf("          int     %d              /*%f               %f*/0,ival,xval,yval);
        }
        printf("0/*                                                           Table 4*/0);
}
```

Appendix 3. Table Generation for Program Segment 5

```
/*      This program repeatedly calls function errfinv() to generate the contents of          */
/*      Table 5.  The output is useable as DSP16 assembly code.                                */

main(argc,argv)
int argc;
char *argv[];
{
        double errfinv();
        int nints;      /*number of intervals */
        int i;
        float xval, xval0,xincr;
        float xmax,ymax;
        float yval;
        float tmp;
        int ival,iscl;
        if(argc == 1)
        {
                printf("needs an integer argument0);
                exit();
        }

        sscanf(argv[1]," %d",&nints);       /*get number of intervals    */
/*      printf(" nints = %d0,nints);        */

        xmax = .4999;           /*not to .5, need to limit yvals to 3.7    */
        ymax = 3.7;
        iscl = 1024*32-1;       /*scale to 2^15-1                          */

        xincr = xmax/nints;     /*set increment                            */
        xval0 = xincr;          /*set starting value                       */
        printf("0);
        printf("                gdevt:0);
        printf("                /*This table is for use with Program Segment 5*/0);
        printf("                                          /*x-value          y-value*/0);
        xval =  0.;
        yval = 0.;
        ival = 0;
        printf("                int     %d                /*%f               %f*/0,ival,xval,yval);
        for(i=0;i<nints; i++)
        {
                xval = xval0 + i*xincr; /*mid point of interval i */
                tmp = errfinv(xval,&yval);
                if(yval > ymax)
                        yval = ymax;
                ival = (yval/ymax)*iscl;
                printf("        int     %d                /*%f               %f*/0,ival,xval,yval);
        }
        printf("0);
        printf("                /*                                           Table 5 */0);
}
```

Appendix 4. C-Source Code for Error Function and Inverse

```
/*  errf     Gaussian or Error function                                          */
/*    This function calculates the Gaussian/Error function and can be used to    */
/*    calculate/tabulate the cumulative distribution function (CDF) of a Gaussian */
/*    random variable.  The function implemented is:                            */
```

$$\mathrm{errf}(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{(\frac{-y^2}{2})}\, dy$$

```
/*    The integral is calculated using the triangle method of integration.  The increment */
/*    used has been experimentally found to be correct to five significant digits.       */
float errf(x)
float x;
{
       double exp(),asin(),sqrt();
       double fac_twopi;
       double xval,xval0;
       double incr;
       double sum,tmp;
       int i;

       /*get multiplicative factor */
       fac_twopi = 4.*asin(1.);
/*     printf("twopi = %f0,fac_twopi);      */
       fac_twopi = 1./sqrt(fac_twopi);

       /*set up counter variables  */
       incr = x/100; /*this works over a large range of values of x    */
       sum = 0.;
       xval0 = incr/2;      /*the mid-point of the intervals           */

       for(i=0;i<100;i++)
       {
              xval = xval0 + incr*i;           /*get next mid-point        */
              tmp = -1. * xval * xval / 2.;    /*argument for exp          */
              tmp = exp(tmp);
              sum = sum + tmp * incr;          /*sum height times width    */
/*            printf("xval = %f  tmp = %f             sum = %f0,xval,tmp,sum);      */

       }
       sum = sum * fac_twopi;
       return(sum);
}


/*  ERRFINV  Inverse of Gaussian or Error Function                                */
/*    This function calculates the inverse of the error function, errf().  This is */
/*    performed by a binary search on the function errf().  The program stops when */
/*    the passed value x is bounded by 0.001.  YVAL is returned as the inverse at X. */
/*    XTST is the true value of X that generates YVAL.                            */
double errfinv(x,y)
double x;
float *y;
{
       float lo, high;
       double xtst;
```

```
        float errf();
        int flag;

        lo=0;
        high = 10;                      /*arbitrarily high    */
        *y = 5;
        flag = 1;
/*      printf("loop %d, xval = %f, lo,y,hi=%f %f %f0,i,xval,
                                        lo, y, high); */
        while(flag)
{
                xtst = errf(*y);
                if( (high - lo) < .001)
                        break;
                if(xtst > x)
                        high = *y;
                else
                        lo = *y;
                *y = (high + lo)/2;
/*              printf("xtst = %f, lo, y, high = %f %f %f0,
                        xtst, lo, *y, high); */
        }
        return(xtst);
}
```

Notes:

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

- ☐ AT&T Microelectronics
  Dept. 50AL330240
  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
  4 Shenton Way #20-00
  Shing Kwan House
  Singapore 0106
  Tel. 225-5233
  Telex RS 42898
  FAX 225-8725

- ☐ AT&T Microelectronics Japan
  7F, Fukoku Seimei Bldg.,
  2-2-2, Uchisaiwai-cho,
  Chiyoda-ku, Tokyo 100 Japan
  Tel. (03) 502-3055
  Telex J32562 ATTIJ
  FAX (03) 593-3307

- ☐ AT&T Microelectronics Spain
  Poligono Industrial de Tres Cantos
  28770 Colmenar Viejo — Madrid
  Spain
  Tel. (34) 1-807-1844
  FAX (34) 1-807-1760

November 1988

AP88-18DMOS

AT&T
The right choice.