SOME EFFICIENT RANDOM NUMBER GENERATORS FOR MICRO COMPUTERS

by

Arne Thesen and Tzyh-Jong Wang

Computer Sciences Technical Report #511

September 1983

SOME EFFICIENT RANDOM NUMBER GENERATORS FOR MICRO COMPUTERS

Arne Thesen and Tzyh-Jong Wang

Departments of Industrial Engineering and Computer Sciences
University of Wisconsin-Madison, Madison, WI 53706

ABSTRACT

August 1983

The relatively slow speed and small word size of the current crop of micro-computers causes the efficient production of pseudo-random numbers on these machines to be considerably more difficult than on larger computers. As a consequence, some micro-computer-based algorithms are excessively time consuming, while other algorithms trade off speed against "randomness". To alleviate this problem we present in this paper several families of pseudo random number generators explicitly designed for use on micro-computers. Some of these are adaptations of well known generators to the micro-computer environment, others are new or lesser known algorithms designed to overcome some of the restrictions intrinsic to the micro-computer's 16 bit environment. For each generator the basic algorithm is discussed and a Pascal implementation is presented. Values of coefficients leading to pseudo random number streams with good statistical properties are recommended and an empirical evaluation of the computational efficiency of the Pascal procedures is offered.

Sponsored by the United States Army under Contract No. DAA29-80-C-0041.

I. INTRODUCTION

The widespread availability of micro-computers has encouraged the implementation of micro-computer based software that previously was only avaliable on larger computers. Frequently this software is adapted from earlier designs for larger computers. In some cases clever design and a heavy use of program and data overlays is all that is needed to "shoe-horn" these large software packages onto smaller computers. In other cases, such as for the class of algorithms discussed here, a fundamental redesign of the underlying algorithms may be required.

The most obvious reason for the failure of pseudo random generators designed for larger computers to work properly on micro-computers is the difference in word size (down from 32 bits to 16 bits). However merely adjusting the algorithm to reflect the reduced word size do not solve the problem. This is because:

1) The limited word size (16 bits) seriously limits the number of unique integers that can be produced using conventional congruential generators.

2) The relative speed of different arithmetic operations is not the same on micro-computers as on larger computers. This is because special purpose arithmetic processors (such as the AMD 8158 or the 8087) are usually not available. Instead, micro-computers usually perform arithmetic by executing a long sequence of compiler generated instructions. This causes integer addition to be an order of magnitude faster than integer multiplication, which in turn is yet another order of magnitude faster than floating point arithmetic.

3) Real numbers are usually represented in a micro-computer with greater precision than integers (4 bytes vs. 2 bytes).

During the research leading to this paper, we investigated the performance of thousands of different procedures and/or coefficients for micro-computer based generation of random numbers. Most looked promising, but failed to perform well when subjected to the tests discussed later in this paper. Those that excelled in these evaluations are presented here.

All programs presented here are written in Pascal/MT+, a widely distributed implementation of Pascal for micro- computers. Most programs are written in Standard Pascal and should be portable to other systems using 16 bit integers, however this claim has not been tested.

II. DESIGN CONSIDERATIONS

Users of pseudo random number generators are concerned with the "randomness" of the numbers generated as well as with the programming and computational effort required to implement it. Many tradeoffs are therefore made when a specific generator is chosen for a given application. In this section we will briefly discuss some of these.

A. Fundamental Properties

The sequence of numbers generated by a pseudo random number generator is not a random sequence, rather, it is a repeating fixed sequence of numbers that pass many reasonable empirical test for "randomness". For example, a simple linear congruential generator may generate the following repeating permutation of the integers 0,1,2,...,14,15:

```
1-14-7-12-13-10-3-8-9-6-15-4-5-2-11-0-1-14-7-12-13-3-8-9-6-15-4-2-11-0-1-14---
<---------------------------------> <-----------------------------> <-----------------
```

The length of the repeating sequence is called the **period** of the generator, and the (ordered) set of numbers generated in a period is referred to as a **cycle**. The **resolution** of a generator is the smallest possible difference between two unequal numbers produced by the generator.

It is desirable to have a generator with as long a period as possible. In addition, it is desirable to have a generator where a cycle contains multiple occurrences of any one integer. (Without this feature, the interval between like numbers in the stream will be equal to the cycle size for all numbers in the stream.) Finally a generator of random integers should have a resolution of one.

B. Computer language requirements

A competent computer programmer is able to implement any pseudo random number generator in almost any higher level computer language. Even so, higher level languages impose several restrictions on the implementation of pseudo random number generators. Among the more common problems are:

1. The value of local variables (such as seeds) might be lost between calls to a procedure.

2. Access to individual bits or bytes of a variable might be difficult.

3. Special purpose arithmetic or logical operations (such as MOD or XOR) might not be available.

4. Parameter passing is frequently time consuming.

In this paper we use Pascal/MT+ to illustrate how these difficulties can be overcome.

The absence of static variables (1. above) is a nuisance but not a serious obstacle. Without static variables, the user must give a global scope to variables never used outside the generator. This increases the probability of coding errors, but it does not increase memory requirements or reduce computing speed. (In Program 2 we show how static variables can be simulated in Pascal/MT+).

Inability to access individual bits or bytes is a more serious problem. Two of our procedures require such access. In Programs 4 and 7 we illustrate how this can be done in Standard Pascal using variant records.

Program 2 requires the use of an "exclusive or" (XOR) operation. This is an operation that usually is not implemented in higher level languages. We chose to implement this function in embedded machine code. While this is a feature supported by Pascal/MT+ and many other Pascal compilers, it is not Standard Pascal, and the resulting code is not universally portable. (However the logic of the programs are clear and recoding of the algorithm should not be difficult.)

C. Statistical Properties

Even though the streams of numbers generated by pseudo random number generators are repeatable and therefore **not** random, our intention is to use these numbers in lieu of truly random numbers. The generated streams must therefore exhibit the same behavior as truly random numbers in the application of interest.

The concept of "randomness" embodies many different statistical properties, and a single statistical test is not available. Instead, different statistical tests are required for different properties. In the research reported here we operationalize the concept of randomness by testing for the following properties:

1. Uniformity of distribution.
2. Randomness of sequence.
3. Absence of autocorrelation.

The specific tests used for each property are discussed in Appendix I. Needless to say all the procedures and coefficients presented in this paper yields pseudo random number streams that pass all of our tests for "randomness".

III. SINGLE BYTE GENERATORS

The byte is the basic information building block in a micro-computer. Two adjacent bytes are used to represent an integer and four adjacent bytes are used to represent a floating point number. As we will show in later sections it is quite practical to construct random deviates of more complex types from a stream of random bytes.

A. Truncated Integers

Most installations have available some procedures for generation of pseudo random integers. It is therefore tempting to develop a procedure that obtains random bytes by first using the higher order and then the lower order byte of this integer. **This is extremely dangerous** as a guarantee of randomness for an integer does not extend to the bytes that make up that integer. In particular the lower order byte of an integer generated using a Linear Congruential (LC) generator (Section IV) is likely to exhibit extremely poor statistical properties. However if the multiplier is carefully chosen, it is possible to obtain a stream of random bytes by returning the higher order byte (only) of a random integer). In Program 1 we present such a procedure:

```
{--------------------}
 function RByte:Byte;
{--------------------}
{  note:   the  following  declarations must appear in  the  main program
   VAR  seed : record case integer of
                         1: (int : integer);
                         2: (lsbyte:byte;
                             msbyte:byte);
                       end;            }
 CONST
     MULT = 1221;  { other good values are: 2837, 3993, 4189, 4293,}
   begin           {                          9237,14789,15125,17245}
     seed.int := MULT * seed.int + 1;
     RByte := seed.msbyte;
     if seed.int < 0 then seed.int:=seed.int+maxint+1;
  end;
```

Program 1: Random byte generator returning the most significant byte of a random integer.

Note that the multipliers recommended for Program 1 are different from the ones recommended for its two byte equivalent (Program 3). We will defer a discussion of LC generators until two byte generators are discussed in the following section. Here we will only point out that:

1) The procedure uses variant records to access the same variable as both an integer variable and as two individual byte variables.
2) The procedure is quite inefficient as one full byte of information is discarded whenever a new byte is generated.
3) The procedure is portable as only Standard Pascal features are used.

B. A Tausworthe Generator

A more efficient approach to the generation of random bytes is to use a procedure proposed by Tausworte[4] that operate directly on bits to form a stream of random bits. This procedure has been shown to produce random number sequences that 1) have improved statistical properties over LC generators, and (2) have an arbitrarly long period independent of the word size of the computer used.

Tausworthe generators are not in widespread use on large computers. This could be because they are difficult to implement efficiently in a higher order language, or because their improved statistical properties are only marginally important on large word size computers. On micro-computers the situation is quite different. Here the improvement in period length and in statistical properties is quite substantial, and, as we shall see, well written Tausworte generators are no more time consuming than other classes of generators.

1. Algorithm

The basic procedure of a Tausworthe type generator is illustrated in Figure 1:

```
            B[i-p]  B[i-r]    B[i]
     {B}   = ..X........X.........X.........
               |        |         ^
               +->XOR<--+         |
                   |              |
                   +--------------+
```

Figure 1: Relationship between bits in a Tausworthe sequence.

Here {B} is defined as a sequence of bits, and the relationship between individual bits in the sequence is defined as:

```
    B[i] = B[i-r]  XOR  B[i-p]

    where :   i    = any integer,
            r , p  = fixed integers with 0<r<p.
              XOR  = the exclusive OR operator yielding 0 if the
                     terms are equal and 1 if they are not.
```

When r and p are properly selected (as primitive trimodals [6]), the maximum period of the stream {B} is 2**p - 1.

2. Implementation

In Program 2 we present an implementation of a Tausworthe algoritmn for the generation of pseudo random bytes. The procedure is an adaptation of an algorithm presented by Lewis and Payne [3]. To avoid the need to access individual bits, the algorithm maintains 8 independent and parallel streams of bits, and the exclusive OR operation is performed on all 8 bits (or one byte) at once. Since each of the eight independent bit streams have a period of 2**p - 1 , the resulting stream of bytes will also have a period of 2**p -1.

```
{---------------------------------------}
 function rbyte(var f,s:integer) : byte;
{---------------------------------------}
 const
   p = 98; pminus1= 97; q = 27;
 type
   BTable = Array [0..98] of char;  { This table holds the queue of bytes to }
 var                                  {operated upon at a  later time}
   B : ^BTable;
 procedure ByteTable;  {This is a programming trick to reserve space for, and initialize }
   begin                  {a  static  table  within   a Pascal/MT+ procedure}
     inline(  9/1/93/191/154/78/5/5/20/189/74/73/179/189/85/182/77/25/14/154/220/195/179/48/178/7/28/
              56/181/80/166/52/209/130/142/151/222/18/241/101/136/137/176/16/148/79/137/155/65/132/174/
              174/90/175/128/112/9/137/172/189/168/137/125/206/70/64/228/237/192/147/16/169/203/240/175/
              239/33/66/13/253/70/162/70/32/160/1/131/239/207/69/63/175/22/196/249/102/224/167/
              0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0/0) ;
   end;
 function xor( first,second:char):byte; { Machine code implementation of XOR}
   var
     temp:byte;
   begin
     inline(
       $3A / first / (* LDA address *)
       $47 /                      (* MOV B A *)
       $3A / second/ (* LDA address *)
       $A8 /                      (* XOR A B *)
       $32 / temp ); (* STA address *)
     xor:=temp;
   end;
 begin
   B  := addr(ByteTable);  { B points to start of  the byte table}
   if f < pminus1 then
     f:=f+1
   else
     f := 0;
   if s <pminus1 then
     s := s+1
   else
     s := 0;
   rbyte := B^[F] ;
   b^[f]:=xor(B^[F],B^[S]) ;
 end;
```

Program 2: Random byte generator using a Tausworthe algorithm.

The table ByteTable contains all entries in the bit streams between the current bits and the ones lagged p positions from the current ones. The pointer F points to the entry lagged p positions as well as to the current entry in the table and the pointer S points to the entry lagged p-q (=r) positions.

3. Remarks

Here we use several unique features of Pascal/MT+. Through the use of the INLINE and ADDR functions we have reserved space (in ByteTable) for a static table of bytes **inside** the generator. This has the advantage that the user need not define the table as a global variable in the main program. The INLINE function is also used to implement the XOR operation. While this function probably could be written in Standard Pascal, its efficiency is greatly enhanced by the use of the 8080 machine code XOR instruction.

IV. INTEGER (2 BYTE) GENERATORS

A. The Linear Congruential Generators.

The linear congruential (LC) algorithm is perhaps the best known and most widely used procedure for computer generation of pseudo random integers. This is not surprising as the algorithm is both easy to understand and easy to code in almost any programming language. Furthermore, for computers with a word size of 32 bits or more, the resulting code is computationally efficient, and it yields random number sequences that pass most reasonable tests for randomness.

For micro-computers we are not so lucky. The smaller word size and slower speed of these computers define limits both on speed and "randomness" that are much less attractive than those found on larger computers. It is therefore essential that designers and users of micro-computer based LC algorithms fully understand how these algorithms work and what their restrictions are. Designers need this knowledge to optimize the performance of their algorithms (apparently trivial changes can quadruple the count of unique numbers generated), and users need this knowledge to decide if an alternative algorithm should be used.

1. Algorithm

The Linear Congruential Generator produces a new random number from the previous one through the following congruential relationship:

```
*-----------------------------------*
| I[i+1] = ( a * I[i] + c ) Mod m   |
*-----------------------------------*
```

```
where I[i]   = the i-th number produced by the generator.
      I[i+1] = the (i+1)-th number produced by the generator.
      a,c,m  = constants.
```

Using this relationship, I[i+1] is computed as the remainder of (a * I[i] + c) divided by m. For example, if a = 13, c=1, m=16 and I[3] = 7 then I[4] is computed as the remainder of (13*7+1)/16 or I[4] = 12.

An important feature of LC generators is the fact that each period contains **at most** one occurence of each of the integers in the range 0 - (**m** - 1). Therefore, the interval between any two like integers is fixed, and, the maximum period is **m**.

The properties of LC generators have been extensively studied. Knuth [2] provides guidelines for selecting the values for the coefficients **a**, **c** and **m** and recommends specific values for large computers. Recommended values for micro-computers are presented later in this section.

2. Implementation

To minimize computational effort, **m** is chosen to be the largest integer that the computer can represent + 1 (2**15 or 32768 on most micro-computer systems). This has the advantage that the Mod operation in the Linear Congruential relationship is performed automatically when the term (a * I[i] + c) is formed.

To minimize the importance of the initial seed, we wish the generator to generate **all** non negative integers less than **m**. It can be shown (Knuth [2]) that this is always possible if **c** and **a** are chosen according to certain rules. Note however that no guarantees are made regarding the statistical properties of the resulting number sequence. One family of generators satisfying these rules are:

```
*------------------------------------*
| I[i+1] :=(a * I[i] +1) Mod 32768   |
*------------------------------------*

   where a = k*4 +1  [k=0,1,2,....]
```

A Pascal implementation of this generator is given in Program 3:

```
{-----------------------------------------------}
 function unif_lcg (var seed : integer) :integer;
{-----------------------------------------------}
   Const
     MULT  =  3993  {recommended  values are: 589, 1813, 2125, 2633, 3993, 4773, 5225}
   begin            {             5737, 5995, 6061, 7149,11097,11245,12217, 20377,25621}
     seed := multiplier * seed + 1;
     if seed  < 0 then seed := seed + maxint +1;
     unif_lcg := seed;
   end;
```

Program 3: Congruential generator

A total of 8192 different values for **a** can be chosen for this generator. Some of these yield "good" sequences, while others yield "bad" sequences. For example, the following sequence is generated if **a** is one:

0-1-2-3-4-5-6-7-8-9-10-11-12-13-14- ....-32766-32767-0-1-2...

This sequence clearly has a period of **m**. However, it will not pass any reasonable test for randomness.

As we are not aware of any method for predicting a priori which values of **a** will result in "good" sequences and which values will result in "bad" sequences, we conducted empirical tests on the output produced by all of the 8192 possible values of **a**. Some of the values of **a** that were found to yield statistically "good" sequences are listed in the program. The reader is cautioned against using other values of the multiplier as most other values were found to fail at least one of the tests discussed in Appx.I.

3. Remarks

A certain degree of simplicity and computational efficiency appears to be gained when **c** is set equal to zero. However, we should point out that generators of the form :

```
*--------------------------------*
|  I[i+1] =(a * I[i])Mod 32768   |
*--------------------------------*
```

have only one fourth of the period of the recommended mixed generator. As a consequence the generator now produces two mutually exclusive sets of (odd) random integers. The actual set produced in any one run depends on the initial seed chosen for that run. Furthermore, as we show in the evaluation section the elimination of the constant **c** has no significant impact on the speed of the resulting procedure.

B. A Tausworthe generator.

In section III we showed that a random byte could be developed from eight parallel random bit streams using a Tausworthe generator. Since this algorithm was not constrained by the word size of the computer used, we can easily extend it to generate a random integer from sixteen parallel streams of random bits. Due to the simplicity of this extension we do not provide a program listing here. However data for the performance of the resulting algorithm is provided in the evaluation section(as Program 4a). Here we chose instead to present an algorithm that illustrates how we can "build" a random integer from two random (Tausworthe) bytes. (This concept is extended in a later section to the construction of floating point numbers.) As shown in Program 4, the key to this algorithm is our ability to use a variant record to represent two adjacent bytes as **both** two individual bytes and a single integer:

```
{----------------------}
 Function Unif:Integer;
{----------------------}
 type
   integertype = Record
     case integer of
       1 : (OneWord : Integer);
       2 : (lsbyte  : Byte    ;
            Msbyte  : Byte    );
     end;
  var
    i:IntegerType;

  { Place Function RBYte right here}

  begin
    i.LSByte := RByte;
    i.MSByte := RByte;
  end;
```

Program 4: Building a random integer from random bytes.

The resulting random number stream has good statistical properties and it has a period of 2**p-1. This period can be substantially extended if the two independent generators have periods that are relative prime (but we have not evaluated how this affects the statistical properties of the resulting random number stream).

C. Shuffle Generators

Shuffle generators combine two or more independent generators to produce a single random number stream with (hopefully) improved statistical properties. Of course this improvement (if any) comes at the cost of reduced computational efficiency. Many elaborate shuffling algorithms have been proposed. The key to success for any of these is the requirement that the driving generators must have periods that are relative prime.

The shuffle algorithmn presented here is adapted from Knuth [2]. As shown in Figure 2, the algorithm maintains an internal table of pseudo random numbers. Whenever a random number is requested, a random index is generated, and the random number stored at the corresponding location in the table is returned. This entry is then replaced by a new entry using the second random number generator.

```
STEP I. Get table index.
      i <-Random index, using generator 1
STEP II.  Get  value stored  at  this location.
      X <- Table[i]
STEP III. Replace this value.
      Table[i] <- Random integer, using generator 2.
```

Figure 2: Knuth's Shuffle Algorithm.

The resulting procedure (Program 5) produces random number streams of quite (Knuth says exceptionally) long periods. Unfortunately this procedure is rather slow due to the fact that we now must perform additional operations to make sure that the two driving generators have periods that are relative prime. It is likely that the procedure can be speeded up by replacing one of the generators by a Tausworte generator such as the one presented in Program 2.However this would reduce the portability of the procedure.

```
{-------------------------------------------------------}
 function shuffle (var table : array [0..127] of integer;
                  var generator : integer ;
                  var selector : integer ;
                      request : integer   { 0 : initialize the table }
                               { 1 : request value output }
                                                 ) : integer ;
{-------------------------------------------------------}
var
     index : integer ;

function DO_GENERATE(var seed:integer ; ): integer  ;
Const
     Multiplier = 5737 {see table 1 for other recommended values}
begin
          seed := seed * multiplier + 1 ;
          if seed < 0 then seed := seed + maxint + 1 ;
         DO_GENERATE := seed ;
     end ;

function DO_SELECT(var  seed:integer ; ): integer  ;
Const
     Multiplier = 6061 {see table 1 for other recommended values}
     var
         valid : boolean ;
     begin
         valid := false ;
         while not valid do
          begin
                 seed := seed * multiplier + 1 ;
                 if seed < 0 then seed := seed + maxint + 1 ;
                 valid := seed < 32749 ; { force period to be
                                  32749-- max prime # < 32768 }
          end ;
         DO_SELECT := hi(seed) ;
     end ;
begin {main}
       case request of
        1 : { request a value }
          begin
           index := DO_SELECT(selector, s_multiplier) ;
           shuffle := table[index] ;
           table[index] := DO_GENERATE(generator,g_multiplier) ;
          end ;
        0 : { initialize the table }
         for index := 0 to 127 do
           table[index] := DO_GENERATE(generator,g_multiplier) ;
        end ; { of case }
end ;{main}
```

Program 5: Sample Pascal procedure for table shuffling.

V. FLOATING POINT GENERATORS

A. Conventional procedures

Real valued random deviates are readily obtained from integer deviates by mode conversion (from integer to real) and by scaling (from 0.0-32767.0 to 0.0-1.0). One such procedure is illustrated in Program 6.

```
var
  seed:integer;
{---------------------}
 function u2125 :real;
{---------------------}
 begin
   seed := 2125 * seed + 1;
   if seed  < 0 then seed := seed + maxint +1;
   u2125 := seed*3.054648E-5;
 end;
```

Program 6: Pascal procedure for uniformly distributed deviates on 0.0-1.0.

This is an acceptable procedure for large computers. However two serious problems restricts its usefulness for micro-computers:

1) Floating point arithmetic is particularly slow on most micro-computers. Program 6 is about 4 times slower than its integer counter part. It would be even slower if we divide by 32,768 instead of multiply by its inverse.

2) Most micro-computer languages use four bytes to store a floating point number while they use only two bytes to store an integer. Program 6 has inherited the cycle and period restrictions of our two byte integer generators.

In the following section we present a new approach to generation of floating point random numbers that overcomes these problems.

B. A Construction Algorithm for Floating Point Deviates.

1. Algorithm

A random variable **u** on [O-1] can be expressed as a function of a random exponent **e** and a random mantissa **m** as follows:

$$u = \frac{m}{M} * 2^{e}$$

where **u** = random variable on [O-1]
e = random variable drawn from the distribution:

$$Pr\{e = i\} = 2^{-(i+1)} \quad i = 0,1,2,..$$

M = a constant
m = Uniformly distributed random variable on [M/2,M).

An algorithm based on this notational convention is presented in Figure 3 (the algorithm is discussed in more detail in [5]):

```
I. INITIAL ASSIGNMENTS

      e  = 0               {Correct value if u > 1/2}
      m  = U(0,M)          {Uniform deviate on 0-M}

II. IS ADDITIONAL WORK REQUIRED?

      If m >= M/2       { this happens half the time}
      then  goto step IV
      else m = m + M/2.

III. ADJUST EXPONENT

   A. Draw random byte(s) until a nonzero byte is found:

      k = RandomByte
      while k = 0    { this happens with a probability of 1/256  }
        e = e - 8
        k = RandomByte.

   B. Scan the byte for the first nonzero bit:

      while k < 128
        k = k  * 2
        e =  e - 1.

IV. RANDOM VARIABLE IS u = f(e,m)
```

Figure 3: Algorithm for generating **u**

The algorithm starts out by assigning a value of zero to the exponent (defining a number on [0.5-1.0]) and a random value to the mantissa. A check is made to see if the resulting mantissa has a valid value (there is a 50% probability that this is true). If so, the algorithm stops as a valid number has been produced.

If the mantissa is not valid (i.e. its first bit is not one), then M/2 is added to **m** to make it valid, and a procedure for generating a random exponent is entered. This procedure is based on the premise that the number of zeroes preceding the first one in a random stream of bits follows the geometric distribution with p = 0.5. Random bytes are drawn until a nonzero byte is found. This byte is then scanned until the first nonzero bit is found. The resulting exponent is computed as the negative value of eight times the number of zero valued bytes plus the number of consecutive zero valued bits in the last byte.

2. Implementation

Our implementation of the algorithm is based on the conventions for representation of floating point numbers adopted by the AMD 9511 (OR INTEL 8231A) hardware floating point unit [1]:

```
  bit 31    30     29 28         25 24 23    16 15       8 7         0
+--------+--------+--+--+--------+--+--+--+--+--+-----+--+-+--------+--+
! sign   ! sign   !   value of      !              Mantissa            !
!  of    !  of    ! of exponent     !most signif!          !least signif!
!mantissa!exponent!  !  !        !  !icant byte !          !icant byte  !
+--------+--------+--+--+--------+--+--+--------+----------+------------+
```

Figure 4: AMD 9511 convention for floating point notation

For numbers between zero and one the sign bit of the mantissa is zero. The sign bit of the exponent is one if the number is less than 0.5 and zero otherwise. In Table 1 we show how the values of **e** and **m** are encoded in the four bytes representing a number in the range 0-1. The correspondence in behavior between **e** and b1 and **m** and b2!b3!b4 is observed.

X	**e**	**m**	b1	b2	b3,b4
1.0 >x>=0.5	0	16,777,215-8,388,608	0	128-255	0-255
0.5 >x>=0.25	-1	16,777,215-8,388,608	127	128-255	0-255
0.25 >x>=0.125	-2	16,777,215-8,388,608	126	128-255	0-255
0.125 >x>=0.0625	-3	16,777,215-8,388,608	125	128-255	0-255
0.0625>x>=0.03125	-4	16,777,215-8,388,608	124	128-255	0-255

Table 1: Values of individual bytes for numbers on [0-1]

Our implementation of an algorithm exploiting this data structure is shown in Program 7.

3. Discussion

While a fixed number of random bytes are always used to generate a random mantissa, the number of bytes **n** used to generate a random exponent is itself a random variable. It can be shown that the distribution of **n** is:

$$P(n=i) = 255 * (1/256)^{\mathbf{i}} /2 \qquad \text{for } \mathbf{i} = 1,2,3,...$$

and the expected value of **n** is 128/255 = 0.5019607.

The procedure is driven by the two independent random byte generators RByte1 and RByte2. In our performance tests we used both the simple truncated integer procedure presented in Program 1, and the more efficient (but not as portable) Tausworthe generator presented in Program 2. When Program 1 was used then Programs 6 and 7 were equally fast (but Program 7 has better resolution and a longer period). When Program 2 was used, then Program 7 was faster than Program 6 and of course it still yielded numbers with better statistical properties.

```
{----------------------}
 function uniform :real;
{----------------------}
 type
   realtype =
     record                    {variant record for byte access}
       case integer of
         1: (unif : real);     {The variable we want}

         2: (exponent : byte; {exponent and sign bits}
             m1 : byte;        {most significant byte of mantissa}
             m2 : byte;
             m3 : byte);       {least significant byte of mantissa}
       end;
 var
   k : byte;
   u : realtype;
 begin
   with u do
     begin
       m1 := RByte1;      {a separate byte generator is assumed}
       m2 := RByte1;
       m3 := RByte1;
       exponent := 0;          { proper value for 0.5 < u  < 1.0    }
       if m1  < 128 then       {by convention the most significant  }
       begin                   {bit 1 in m1 must be 1               }
         m1 := m1 + 128;
         exponent := 127;
         k := RByte2;
         While k = 0 do
           begin
             exponent := exponent - 8;
             k := RByte2;
           end;
         if k < 128 then begin
           if k >= 64 then exponent := exponent -1
             else if k >= 32 then exponent := exponent -2
               else if k >= 16 then exponent := exponent - 3
                 else if k >= 8 then exponent := exponent -4
                   else if k >= 4 then exponent := exponent - 5
                     else if k >= 2 then exponent := exponent -6
                       else exponent := exponent - 7
        end;
       uniform := unif;
     end;
   end;
```

Program 7: Pascal/MT+ Implementation of uniform random number
           generator using AMD 9511 floating point notation.

VI. EVALUATION

All the generators presented here were subjected to extensive performance tests. This included statistical tests for distribution, sequence and autocorrelation as well as timing tests for computational efficiency. The statistical tests used for this purpose are summarized in Appendix I. All generators presented here passed all statistical tests. A further discussion of statistical test results is therefore omitted. The results of the timing tests differed substantially for the different generators. These are summarized here together with other important intrinsic performance characteristics.

The timing data presented below was obtained by measuring the time required to generate 32,000 random numbers on a Sierra Data Sciences single board computer running at 4mz under the Turbodos operating system (a CP/M dialect).

The reader is warned against reading too much into minor differences in execution times. Such differences are as likely to be caused by differences in programming styles and data transfer methods as by intrinsic algorithmic performance differences. For example we found that the Tausworthe byte generator (Program 2) performed three to four seconds faster in our tests when the resulting byte was maintained as a global variable rather than returned through the function.

A. Byte generators

The performance of our two byte generators is summarized in Table 2. While the Tausworthe generator was slightly slower than the LC generator, we have confirmed the fact that this difference is primarily due to differences in data transfer methods and not due to differences in algorithmic design. The most important difference between the generators is therefore the fact that the Tausworte generator has a substantially longer period than the LC generator. However this improvement in performance is gained at the cost of not using Standard Pascal.

PRO-GRAM	GENERATOR	RANGE	APPROX. PERIOD	RESO-LUTION	INTERVAL BETWEEN LIKE NUMBERS	ARGUMENT PASSED	RELATI SPEED	STANDARD PASCAL
1	Truncated integ	0-255	2**15	1	random	no	12"	yes
2	Tausworthe byte	0-255	2**98	1	random	yes	16"	no

Table 2: Relative Performance of Random Byte Generators.

B. Integer generators

The performance of our integer generators is summarized in Table 3. The standard LC generator (Program 3) performed faster than any of the other generators. However the period for this generator (32768) is so short that we hesitate to recommend its use in lengthy simulations. Program 3a is a standard LC generator with **c** set equal to zero. Data for this generator is included to illustrate the fact that the omission of **c** reduces the period of a generator without improving its speed. The two Tausworthe generators were slower than the LC generators. Again we verified the fact that the speed difference between these two algorithms was only caused by differences in data transfer methods. Finally the shuffle algorithm was about three times slower than the LC generator. However this generator has a substantially longer period and is written in Standard Pascal, it might be the generator of choice for users without access to Pascal/MT+.

PRO-GRAM	GENERATOR	RANGE	APPROX. PERIOD	RESO-LUTION	INTERVAL BETWEEN LIKE NUMBERS	LOW ORDER BYTE "RANDOM"	RELATI SPEED	STANDARD PASCAL
3	LCG (I=a*I+1)	0-32767	2**15	1	2**15	no	15"	yes
3a	K=2483*K	1-32767	2**13	4	2**13	no	15"	yes
4a	Tausw. 2 bytes	0-32767	2**96	1	random	yes	31"	no
4	Tausw. Integer	0-32767	2**97	1	random	yes	23"	no
5	TABLE SHUFFLE	0-32767	2**30	1	random	yes	45"	yes

Table 3: Relative Performance of Integer Generators.

C. Floating Point Generators

The performance of our floating point generators is summarized in Table 4. We note that the fastest generator is our implementation of the construction algorithm using the nonstandard Tausworthe byte generator (Program 7a). However the implementation of this procedure using Standard Pascal (Program 7) was as fast as the conventional LC based generator and it exhibited substantially better statistical properties.

PRO-GRAM	GENERATOR	RANGE	APPROX. PERIOD	RESO-LUTION	INTERVAL BETWEEN LIKE NUMBERS	LOW ORDER BYTE "RANDOM"	RELATI SPEED	STANDARD PASCAL
6	U= I * 0.0000306	.0000 - 0.9999694	32767	2**-15 max	2**15	no	57"	yes
6a	U=I/32676	.0000 - 0.9999694	32767	2**-15 max	2**15		72"	yes
7	U=F(e,m)+Prog1	.0 - 1.0	>>2**36	2**-23 max	random	yes	57"	yes
7a	U=F(e,m)+Prog 2	.0 - 1.0	>>2**36	2**-23 max	random	yes	45"	no

Table 4: Performance of Floating Point Generators.

VII. SUMMARY

This paper presents the findings of an extensive evaluation of thousands of different combinations of algorithms and constants for micro-computer based random number generators. All of the generators presented here have been shown to pass reasonable tests for uniformity of distribution, randomness of sequence and absence of auto correlation. In addition each generator dominate the others in at least one of the dimensions of speed, period and portability.

We had expected to observe a tradeoff between speed and "randomness" , however no such relationship was observed. In fact the fastest generator of floating point numbers also exhibited the longest period. We also had expected to observe a strong relationship between programming style and computational efficiency. This expectation was confirmed. Clever use of nonstandard language features do improve speed. Likewise the manner in which data is transferred to and from the procedures has a significant (15% up to 30%) effect on relative performance.

However, we feel that the most important lesson to be learned from this study is the fact that we were completely unable to predict in advance whether or not a given algorithm would exhibit good or bad statistical properties. The likelihood of improving an algorith through ad hoc changes are slim at best.

VIII. BIBLIOGRAPHY

1. Intel Corp., **8231A Arithmetic Processing Unit,** Preliminary Data Sheet, 1981.

2. Knuth, **The Art of Computer Programming.** Addison-Wesley, 1969.

3. Lewis, T.G. and Payne W.H. **Generalized Feedback Register Pseudorandom Number Generator.** JACM vol.20, No 3. July 1973. pp.456-468.

4. Tausworte, R.C., **Random Numbers Generated by Linear Recurrence, Modulo Two,** Math. Comput. 19 (1965) 201-209.

5. Thesen, Arne., **An Efficient Generator of Uniformly Distributed Random Deviates Between Zero and One.** Technical Report. Mathematics Research Center, University of Wisconsin-Madison, 1983.

6. Zierler, Niel and John Brillhart, **On Primitive Trinomials (Mod 2),** Information and Control, Vol 13 pp 541-554. 1968.

APPENDIX I: STATISTICAL TESTS.

All procedures listed in this paper, when used with recommended parameter values, produce streams of numbers that will pass the following statistical tests. (Streams produced using other parameter values are likely to fail these tests.)

A. Distribution.

The range of numbers produced by a generator is divided into 128 equal subranges. A stream of 4100 numbers is then generated and a frequency count of observed values in each subrange is developed. These empirical counts are then compared to the expected counts (4100/128). A two sided **Chi-square** test is used to test the hypothesis that the observed errors in each interval are reasonable errors that could have occurred if the original data were drawn form the uniform distribution.

This procedure was replicated ten times. All generators presented here failed this test at most once on the 95% level.

B. Sequence.

A **run test** was used to test the hypothesis that the numbers generated for the above distribution test appeared in random order. Specifically we accepted the hypothesis that a generator produced integers in a random sequence if no more than one of the ten streams failed the following test on the 95% level.

1. Define a run of length n to be a sequence such that:

   x[1] < x[2] ... x[n] > x[n+1]

   The sequence 89-456-893-05 is a run of length three.

2. Scan the set of numbers sequentially to determine the length of individual runs. (Discard the run terminator x[n+1], do not include it in the next run!) The sequence 234-564-234-453-789-990-78 has two runs; one of length two (234-564-234) and one of length three (453-789-990-78).

3. Develop a 6-bin histogram counting the number of runs of length 1, 2, 3, 4, ,5 and 6 and more.

4. Use a chi-square test to compare the observed distribution of run lengths to the expected distribution:

Run length	1	2	3	4	5	6 or more
Probability	1/2	1/3	1/8	1/30	1/144	1/720

C. Autocorrelation

A good pseudo random number generator should have the feature that the value of the next number to be generated can not be predicted from the values of previously generated numbers. If this feature is present then we cam consider the random number stream to be a stationary time series with a constant mean and variance, and a unit autocorrelation matrix.

In our test for the absence of auto correlation, we estimated the autocorrelation and partial auto correlation matrices (for the first 128 lags) for the ten streams generated for the distribution test above.

The hypothesis of no autocorrelation was accepted if no systematic patterns such as autoregressive or moving average were observed from the error adjusted correlogram for any of the streams evaluated.