*Behavior Research Methods, Instruments, & Computers*
*1987, 19 (4), 397-399*

# Cautions regarding random number generation on the Apple II

JAMES W. ALDRIDGE
*Pan American University, Edinburg, Texas*

Undocumented characteristics of the pseudorandom number generators in Applesoft BASIC and Apple Pascal are described that cause identical sequences to be generated on different executions of programs written in those languages. Although it is relatively easy for this problem to escape notice, in both cases the problem is easily corrected once its existence is known.

The purpose of this paper is to call attention to undocumented characteristics of the pseudorandom number generators provided with Applesoft BASIC and Apple Pascal. It is relatively easy to inadvertently use the generators in such a way that identical sequences are generated on different executions of an experiment-controlling program, introducing spurious effects into the data. It is also relatively easy for this to occur without being noticed by the experimenter.

Pseudorandom number generators in general require a "seed," a number on which to base a pseudorandom number sequence. In the Apple II series of computers, a potential seed is available in memory locations $4E and $4F hexadecimal (78 and 79 decimal). The monitor routine KEYIN reads the keyboard by continuously looping until a key is pressed, with each loop incrementing the value in $4E. Each time the value in this location rolls through $FF back to 0, $4F is incremented. This results in the values in $4E and $4F representing a number between 0 and $FFFF (0 and 65535). Since the entire process counts to $FFFF and starts again at 0 in less than a second, the value at any instant is quite arbitrary as long as there has been some keyboard input. There will have been some keyboard input in any situation except when one uses a "turnkey" program, which automatically runs as soon as the machine is powered on.

The keyboard input need not be required by a particular program. As soon as the power-on routines have finished operating, the monitor routine begins waiting for input and incrementing the number, which will be unaffected by subsequent loading of individual programs or of many high-level languages. Thus, if a program even requires a keyboard command to be executed, the value in $4E and $4F constitutes a seed suitable for most applications. Problems arise in the manner in which higher-level languages use (or do not use) the seed.

Preparation of this paper was supported by grants BNS-7919801 from the National Science Foundation and SO6RR08038 from the National Institute of Mental Health/MBRS. Reprints may be obtained from James W. Aldridge, Psychology Department, Pan American University, Edinburg, TX 78539.

## APPLESOFT BASIC

The language built into all Apple II series computers since the Apple II+ is Applesoft BASIC, which provides the function RND for generating pseudorandom sequences. RND takes an argument $n$, and returns a pseudorandom number greater than or equal to 0, but less than 1. This value may be converted into an integer within any range from 0 to $x-1$ by multiplying it by $x$ and rounding down.

The behavior of RND is governed by the value of the argument $n$. If $n$ is negative, it acts as a seed so that a pseudorandom sequence is begun that will be restarted whenever RND is again used with that particular negative argument. If $n=0$, the most recently generated number is returned. These features can be convenient in debugging and for saving storage space by saving a seed rather than an entire sequence. The problem arises when the programmer intends to generate new pseudorandom numbers using a positive argument.

The Applesoft reference manual (Apple Computer, 1981) cites only the following about using positive arguments with the RND function: "If aexpr is greater than zero, RND(aexpr) generates a new random number each time it is used" (p. 102), and "Every time RND is used with any positive argument, a *new* random number from 0 to 1 is generated, unless it is part of a sequence of random numbers initiated by a negative argument" (p. 159). A reasonable assumption is that RND with a positive argument generates a value based upon the value in locations $4E and $4F discussed above. This is not the case.

Applesoft in fact makes no use of the $4E-$4F value. What actually happens is that RND with any positive argument or series of positive arguments generates exactly the same sequence each time the machine is powered on. This is not easily noticed, however, since the sequence is begun only when the machine is activated. The function thus appears to be generating a new sequence every time a program is rerun, even if the machine has been rebooted. It is only when a machine is powered off and back on that the sequence begins repeating itself. Furthermore, since the values generated by RND normally will be transformed differently by each user program, the repetition at power-on will usually be noticed only if the same program is used immediately after each power-on.

The potentially unfortunate consequences of this undocumented "feature" are illustrated by the following study. In a memory experiment, each subject was supposed to have been presented an individually randomized list of words. After testing a large number of subjects, the research assistant gathering the data happened to notice that one particular ordering of the list seemed to be reappearing with some frequency. This had not been noticed earlier because presentation of the lists and response scoring were done by the computer, and the problem could have easily escaped notice.

We discovered that the subjects receiving the repeated lists were those tested at the beginning of each day, immediately after the computer had been turned on. An Apple representative was contacted about the problem, and we were advised to insert a statement of the form X=RND (−1*(PEEK(78)+256*PEEK(79))) after keyboard input but before the first use of RND in the original program.

This statement converts the $4E-$4F value to a negative integer, and then uses it as a negative argument for RND. This begins a new random number sequence, which is different for each value in $4E-$4F. It is necessary to insert the statement only once in a program, after keyboard input but before the first use of RND. As long as there has been keyboard input of any kind between power-on and the occurrence of the statement, this solution is acceptable for most applications in experimental psychology. There are, however, other limitations to RND (Modianos, Scott, & Cornwell, 1987) that are of concern in the development of simulations or other applications requiring relatively long series of nonrepeating pseudorandom numbers.

## APPLE PASCAL

There also exists an undocumented problem with random number generation in Apple Pascal. The problem arises from a different source than that discussed above, but has a similar consequence. Apple Pascal is a variant of UCSD Pascal, which provides the capability for running very large programs with the use of "overlaying." Large programs may be divided into segments, with each segment resident in memory only while it is actually in use. In addition, routines used by many programs may be stored in libraries in the form of units that are read into memory as necessary.

Apple provides two pseudorandom number routines in the unit APPLESTUFF. One of these routines, the function RANDOM, returns a pseudorandom number between 0 and 32767. The other routine, the procedure RANDOMIZE, functions in a manner similar to that of the Applesoft RND solution discussed above. RANDOMIZE seeds RANDOM with a time-dependent value determined by input and output, and in most cases avoids the problems produced by Applesoft's RND.

The problem with Pascal's routine occurs only in one specific situation and may therefore be easily overlooked. The manner in which units and segments are loaded and swapped may be controlled by commands to the compiler that are embedded in a program's source text. Under default conditions, certain types of units (intrinsic units) are loaded into memory as soon as the calling program is executed, and remain in memory for the duration of the calling program. If, however, the compiler NOLOAD option is used, intrinsic units are loaded into memory only at the time they are called and are kept in memory only as long as they are active. This option allows for extremely efficient use of available memory, but at the cost of execution speed. With NOLOAD in effect, every use of a function or procedure in an intrinsic unit requires reloading of the unit from the system disk.

The pseudorandom number problem in Apple Pascal results from the incorporation of RANDOMIZE in APPLESTUFF, which is an intrinsic unit. There appears to be no problem unless NOLOAD is in effect—one use of RANDOMIZE properly seeds a sequence used by all subsequent calls to RANDOM. If, however, NOLOAD is in effect, the effect of RANDOMIZE is negated and RANDOM will always return the same sequence of values.

The problem appears to result from the swapping of the unit caused by NOLOAD. Consider the following procedure:

```
BEGIN RANDOMIZE; WRITE(RANDOM) END;
```

With NOLOAD this sequence always writes the same value, whereas without NOLOAD the value will be different at each execution. This presumably results from the fact that when NOLOAD is in effect, the statement RANDOMIZE causes the loading of APPLESTUFF, the execution of RANDOMIZE, and the release of APPLESTUFF from memory. The release of APPLESTUFF also apparently releases the variable containing the seed value created by RANDOMIZE. The call to RANDOM then causes a new loading of APPLESTUFF, as though it had never been previously loaded, and RANDOM generates a value as though the previous RANDOMIZE had never been executed.

Once the problem is noticed, the solution is relatively simple. Another compiler option, RESIDENT, prevents the automatic swapping of segments or the swapping of intrinsic units that would otherwise be caused by NOLOAD. The effect of RESIDENT is restricted to the procedure that contains the option and to units and segments specified in the option. Thus, the statement

```
BEGIN {$R APPLESTUFF} RANDOMIZE;
WRITE(RANDOM) END;
```

functions properly even with NOLOAD in effect. The RESIDENT specification ($R) prevents the swapping out of APPLESTUFF between the calls to RANDOMIZE and to RANDOM, without affecting the action of NOLOAD on portions of the program outside of the procedure. Since the option is specific to APPLESTUFF, it also would not affect the action of NOLOAD on any other unit calls added to the procedure. Further information is available in the language reference manual provided with Apple Pascal (Apple Computer, 1980), and Wichmann and Hill (1987) reported their own Pascal random number function that produces very long nonrepeating sequences.

It should be stressed that the above problems are not merely interesting quirks that will rarely be encountered in actual programming situations. The nature of the problems is such that they can have disastrous consequences for the integrity of one's data, but can easily escape notice. It was only by chance that my assistant noticed the problem with Applesoft in the above example, due to the restricted condition under which it occurs. With Pascal, it is often the case that a program will be completed and tested, and at a later time some change in conditions requires addition of the NOLOAD option. A program might not be extensively retested after so minor a change, and an experimenter may fail to notice that number sequences were no longer being properly generated. It is even reasonable to suspect that data already in the literature have been contaminated by these undocumented problems.

## REFERENCES

Apple Computer, Inc. (1980). *Apple II Apple Pascal language reference manual*. Cupertino, CA: Author.

Apple Computer, Inc. (1981). *Applesoft II BASIC programming reference manual*. Cupertino, CA: Author.

Modianos, D. T., Scott, R. C., & Cornwell, L. W. (1987). Testing intrinsic random-number generators. *Byte*, **12**, 175-178.

Wichmann, B., & Hill, D. (1987). Building a random-number generator. *Byte*, **12**, 127-128.

(Manuscript received March 19, 1987;
revision accepted for publication May 8, 1987.)