BY JEFF HOLTZMAN
COMPUTER EDITOR

Generating Random Numbers

DID YOU EVER THINK ABOUT DEVELOPING A FOOLPROOF METHOD OF WINNING THE STATE LOTTERY? AT THE HEART OF YOUR SYSTEM YOU PROBABLY HAD A RANDOM-NUMBER GENERATOR—PREFERABLY A PORTABLE ONE. DOESN'T SOUND THAT hard: Just slap together a little 16-character LCD, a microcontroller, and a way to generate random numbers. It couldn't be that difficult, could it? Guess again.

I'll tell you right now that hardware is your best bet. Nonetheless, for the sheer pleasure of it, we're going to beat our heads against the wall and see what we come up with software-wise. If that doesn't convince you, I've got 200 pages of dense mathematics for you to read.

What Is A Random Number?

Glad you asked. Actually, a number is just a number; there's no such thing as a random number. What's important is to be able to generate sequences of unrelated or apparently unrelated numbers. That "apparently unrelated" is the catch. It implies some sort of relationship between the generator and the user. The point is that even if a sequence is not random, if the user thinks it is, that's good enough.

People who are used to programming in high-level languages typically use a function like one of those shown in Table 1 to generate random numbers. Typically, you call the seed function first with some "random" value that kicks things somewhere in the middle of the sequence—hopefully differently from the last run.

Note that there is no assembly-language function. Life would be easier for lots of people if CPU designers incorporated a read-only "random" register that ripped along at some multiple of the basic clock frequency.

Of course, those high-level functions just call underlying library code, which is what does the work. People who analyze the output of library generators typically find them lacking, so be warned. If you need high-quality random number sequences, look elsewhere.

There are many methods for generating sequences of random numbers. The most popular is called the linear congruential method, which was invented by D. H. Lehmer in 1949. The chief characteristics of the Lehmer method are that it is simple, deceptive, insidious, and maddening.

The Lehmer Generator

The basic formula is simply:

$$x(i+1) = (a * x(i) + c) \bmod m$$

LISTING 1—QBASIC RANDOM-NUMBER GENERATOR

```
REM A multiplicative congruential random-number or Lehmer generator
REM basic formula: x(i+1) = (a*x(i) + c) mod m
REM from C User's Journal, June 1995, Jerry Dwyer
REM jkh, 2/28/98

DIM c AS INTEGER
DIM m AS INTEGER
DIM i AS INTEGER

DIM Count AS INTEGER
DIM Seed AS INTEGER

DIM x AS INTEGER
DIM y AS INTEGER

a = 5
c = 0
m = 7

Count = 100
Seed = 1

PRINT "-------------------------"
x = Seed
FOR i = 1 TO Count
    y = (a * x + c) MOD m
    PRINT x;
    x = y
NEXT i
```

LISTING 2—LEHMER GENERATOR IN C

```
/*
** longrand() — generate 2**31-2 random numbers
**
** public domain by Ray Gardner
**
** based on "Random Number Generators: Good Ones Are Hard to Find",
** S.K. Park and K.W. Miller, Communications of the ACM 31:10 (Oct 1988),
** and "Two Fast Implementations of the 'Minimal Standard' Random
** Number Generator", David G. Carta, Comm. ACM 33, 1 (Jan 1990), p. 87-88
**
** linear congruential generator f(z) = 16807 z mod (2 ** 31 - 1)
**
** uses L. Schrage's method to avoid overflow problems
*/

#define a 16807         /* multiplier */
#define m 2147483647L   /* 2**31 - 1 */
#define q 127773L       /* m div a */
#define r 2836          /* m mod a */

long nextlongrand(long seed)
{
    unsigned long lo, hi;

    lo = a * (long)(seed & 0xFFFF);
    hi = a * (long)((unsigned long)seed >> 16);
    lo += (hi & 0x7FFF) << 16;
    if (lo > m)
    {
        lo &= m;
        ++lo;
    }
    lo += hi >> 15;
    if (lo > m)
    {
        lo &= m;
        ++lo;
    }
    return (long)lo;
}

static long randomnum = 1;

long longrand(void)             /* return next random long */
{
    randomnum = nextlongrand(randomnum);
    return randomnum;
}

void slongrand(unsigned long seed)      /* to seed it */
{
    randomnum = seed ? (seed & m) : 1;  /* nonzero seed */
}

#ifdef TEST

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    long reps, k, num;
    unsigned long seed;

    reps = 10000;
    seed = 1;

    /*
    ** correctness test: after 10000 reps starting with seed 1,
    ** result should be 1043618065
    */

    if (argc > 1)
        reps = atol(argv[1]);
    if (argc > 2)
        seed = atol(argv[2]);

    printf("seed %ld for %ld reps...\n", seed, reps);
    slongrand(seed);
    for (k = 0; k < reps; ++k)
        num = longrand();
    printf("%ld\n", num);

    return 0;
}

#endif
```

In words, to get the next value, take the current value, multiply it by a, add c to it, divide by m and take the remainder. Choosing good values for a, c, and m is not simple. Bad values quickly degenerate into non-random sequences. Listing 1 shows a little QBasic program to play with.

For short runs, the algorithm is worthless, because it repeats quickly. For example, suppose you wanted to create an electronic dice game, producing random numbers between 1 and 6. The QBasic program in Listing 1 produces the following results:

```
1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6
2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5
4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3
1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6
2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5
4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3
1 5 4 6
```

You probably wouldn't want to create a dice game based on that.

For best results, the value of m should be the data-bus width of your CPU ±1. On a 32-bit CPU, a good value is 2**32-1 (= 2,147,483,647), which happens to be a prime number. (By the way, m and a should be relatively prime.) A good value for a is 16807. Unfortunately, you can't use the value 2147483647 in QBasic (it's 16-bit). You can use 32767, which seems to work pretty well.

The C code in Listing 2 is a 32-bit variant that codes around potential overflow problems due to the use of signed integer arithmetic. Obviously that's not a problem on any modern computer, but it's a different story on, say, an 8-bit microcontroller.

Also Keep In Mind

As you think about the problem, here are a few other things to keep in mind:

- There are other algorithms, but they take lots of code.
- Verifying randomness and distribution are not trivial tasks. A degree in statistics would help.
- You can't simply feed the output of one random-number-generator routine to the input of another to increase randomness.

For some background and algorithms, see the 6/95, 6/96, and 8/96 issues of *C/C++ User Journal*. For a comprehensive treatment, the new version (Vol. 2, 1998) of Donald Knuth's *The Art of Computer Programming* has more than

TABLE 1—RANDOM NUMBER FUNCTIONS IN VARIOUS LANGUAGES

Language	Seed Function	Generator Function	Returns
MS QBasic	Randomize	Rnd	Single precision value 0.0 < n < 1.0
ANSI C	srand	rand	integer 0 <= n <= RAND_MAX
Java	srandom	random	single 0.0 < n < 1.0
Pascal (Borland Delphi)	Randomize	Random (range)	single, 0.0 <= n < 1.0, or integer 0 <= n < range

you'll ever want to know (nearly 200 pages) about random-number generation. He suggests using Chapter 3 as the basis of a semester course for college senior or grad students. He doesn't even acknowledge the existence of CPUs of less than 16 bits.

I don't believe there is an efficient random number generator for an 8-bit CPU. Depending on your needs, you'll probably have to do double- or quad-precision adds and multiplies.

Hardware Solutions

Perhaps the easiest way is to set a timer to fire at a rapid rate relative to the tasks done by your application. Then, each time it fires, increment the count in a register. It's like the "tick" count in a PC. For that to work, your usage will occur at random intervals relative to the clock (*e.g.*, user events such as pressing a switch). Of course, it assumes that you have a timer, and that you're not using it for anything else. If you are, multiplex the timer interrupt and maintain separate counts.

Registers and RAM are typically uninitialized (*i.e.*, they have "random" values). Use one such value to seed the timer. In this case, the timer would run at a different rate every time you boot.

Conclusions

Do you have an algorithm for generating pseudo-random number sequences on an 8-bit microcontroller? Are you confident that it works? If you can answer yes to both questions, send in your method. I'll run it through some tests, and if it survives, I'll publish the results here. See you next time; stay in touch via e-mail at jeff@ingeninc.com. EN

EQUIPMENT REPORT

continued from page 14

the right command set—it then tells you what the proper code is. To top it off, the remote even has a built-in light that illuminates the buttons—you might want to use it to replace your present remote control.

Setup

The Home Director Starter Kit is very easy to set up. First you have to install four AAA batteries in the remote and two AAA batteries in the PC Connection Module—those two batteries preserve the routines stored in the module's memory even in a power outage. The PC Connection Module must be plugged into an outlet near your computer. An included cable connects the module to the serial port on the PC. A light or appliance that you wish to control is then plugged into the PC Connection Module. The Lamp Module and Remote Module can be plugged into outlets anywhere in your home; their specific locations are determined by what items you want to control and where those devices are located.

Next you have to install the software that is provided on a CD-ROM. The software presents a simple graphical interface that lets you install and configure modules in the system. Each module and its functions are graphically depicted in the software interface, and a right mouse click brings up various options for each module. All of the lights and appliances in the system can be controlled directly from the PC, and at this point you can test any of the attached devices.

Included with the software are a couple of sample routines that you can test out. One routine turns on a lamp at 50% brightness, immediately turns on an appliance, turns the lamp up to 100% brightness after five minutes, turns off the appliance after 10 minutes, and then turns off the lamp after 15 minutes. You can write your own routines or edit the supplied routines according to your own needs. You can give each routine a unique name and even label the modules according to the appliances you have plugged into them. You might want to define wake-up and goodnight routines, or perhaps weekday and weekend routines.

The routines allow you to tailor your lighting and appliance scheduling for each day of the week. To save time, the software lets you duplicate and drag-and-drop parts of one routine into another routine. Because a PC keeps track of the date and time, advanced functions let you make custom routines for specific dates or even configure dawn and dusk routines once you have entered your time zone—the software then knows when the Sun comes up and when it goes down where you live, and it even accounts for Daylight Savings Time.

IBM's Home Director Starter Kit is just the ticket for people who would like to automate the lights and appliances in their home without spending a lot of money on equipment or a lot of time on installation. And with readily available expansion modules, the system is fully customizable. If you already have a PC, you already own the most expensive part of the system. Now all you need is the Home Director. For more information, contact IBM (Route 100, Somers, NY 10589; Tel: 800-426-7235, Ext.4340; Web: www.pc.ibm.com/ homedirector) directly, or circle 15 on the Free Information Card. EN