```
        if (n != nold) {                  If n has changed, then compute useful quanti-
            en=n;                             ties.
            oldg=gammln(en+1.0);
            nold=n;
        } if (p != pold) {                If p has changed, then compute useful quanti-
            pc=1.0-p;                         ties.
            plog=log(p);
            pclog=log(pc);
            pold=p;
        }
        sq=sqrt(2.0*am*pc);               The following code should by now seem familiar:
        do {                                  rejection method with a Lorentzian compar-
            do {                              ison function.
                angle=PI*ran1(idum);
                y=tan(angle);
                em=sq*y+am;
            } while (em < 0.0 || em >= (en+1.0));       Reject.
            em=floor(em);                 Trick for integer-valued distribution.
            t=1.2*sq*(1.0+y*y)*exp(oldg-gammln(em+1.0)
                -gammln(en-em+1.0)+em*plog+(en-em)*pclog);
        } while (ran1(idum) > t);         Reject. This happens about 1.5 times per devi-
        bnl=em;                               ate, on average.
    }
    if (p != pp) bnl=n-bnl;               Remember to undo the symmetry transforma-
    return bnl;                               tion.
}
```

See Devroye [2] and Bratley [3] for many additional algorithms.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 120ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §X.4. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3].

7.4 Generation of Random Bits

The C language gives you useful access to some machine-level bitwise operations such as << (left shift). This section will show you how to put such abilities to good use.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their high-order bit (i.e., test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special-purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on "primitive polynomials modulo 2." The theory of these polynomials is beyond our scope (although §7.7 and §20.3 will give you small tastes of it). Here, suffice it to say that there are special polynomials among those whose coefficients are zero or one. An example is

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

which we can abbreviate by just writing the nonzero powers of x, e.g.,

$$(18, 5, 2, 1, 0)$$

Every primitive polynomial modulo 2 of order n (=18 above) defines a recurrence relation for obtaining a new random bit from the n preceding ones. The recurrence relation is guaranteed to produce a sequence of maximal length, i.e., cycle through all possible sequences of n bits (except all zeros) before it repeats. Therefore one can seed the sequence with any initial bit pattern (except all zeros), and get $2^n - 1$ random bits before the sequence repeats.

Let the bits be numbered from 1 (most recently generated) through n (generated n steps ago), and denoted $a_1, a_2, \dots, a_n$. We want to give a formula for a new bit a_0. After generating a_0 we will shift all the bits by one, so that the old a_n is finally lost, and the new a_0 becomes a_1. We then apply the formula again, and so on.

"Method I" is the easiest to implement in hardware, requiring only a single shift register n bits long and a few XOR ("exclusive or" or bit addition mod 2) gates, the operation denoted in C by "$\wedge$". For the primitive polynomial given above, the recurrence formula is

$$a_0 = a_{18} \wedge a_5 \wedge a_2 \wedge a_1 \tag{7.4.2}$$

The terms that are $\wedge$'d together can be thought of as "taps" on the shift register, $\wedge$'d into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

While it is simple in hardware, Method I is somewhat cumbersome in C, because the individual bits must be collected by a sequence of full-word masks:

```
int irbit1(unsigned long *iseed)
```
Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is modified for the next call).

```
{
    unsigned long newbit;                   The accumulated XOR's.

    newbit = (*iseed >> 17) & 1             Get bit 18.
        ^ (*iseed >> 4) & 1                 XOR with bit 5.
        ^ (*iseed >> 1) & 1                 XOR with bit 2.
        ^ (*iseed & 1);                     XOR with bit 1.
    *iseed=(*iseed << 1) | newbit;          Leftshift the seed and put the result of the
    return (int) newbit;                        XOR's in its bit 1.
}
```

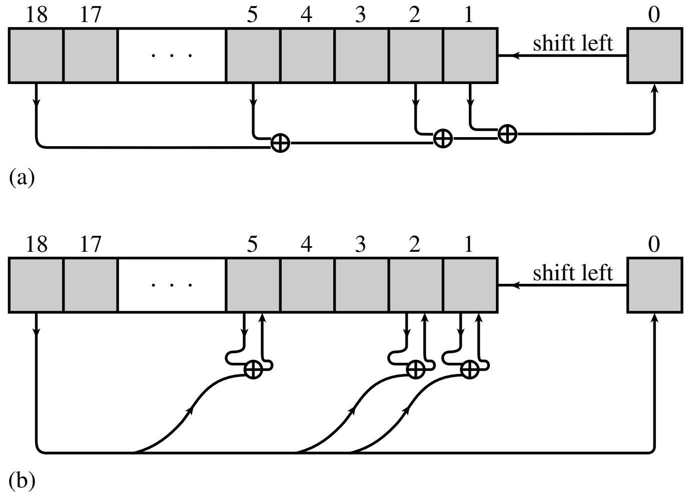

Figure 7.4.1. Two related methods for obtaining random bits from a shift register and a primitive polynomial modulo 2. (a) The contents of selected taps are combined by exclusive-or (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. (b) Selected bits are modified by exclusive-or with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software.

"Method II" is less suited to direct hardware implementation (though still possible), but is beautifully suited to C. It modifies more than one bit among the saved n bits as each new bit is generated (Figure 7.4.1). It generates the maximal length sequence, but not in the same order as Method I. The prescription for the primitive polynomial (7.4.1) is:

$$
\begin{aligned}
a_0 &= a_{18} \\
a_5 &= a_5 \wedge a_0 \\
a_2 &= a_2 \wedge a_0 \\
a_1 &= a_1 \wedge a_0
\end{aligned}
\tag{7.4.3}
$$

In general there will be an exclusive-or for each nonzero term in the primitive polynomial except 0 and n. The nice feature about Method II is that all the exclusive-or's can usually be done as a single full-word exclusive-or operation:

```
#define IB1 1                     Powers of 2.
#define IB2 2
#define IB5 16
#define IB18 131072
#define MASK (IB1+IB2+IB5)

int irbit2(unsigned long *iseed)
Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which is
modified for the next call).
{
    if (*iseed & IB18) {          Change all masked bits, shift, and put 1 into bit 1.
        *iseed=((*iseed ^ MASK) << 1) | IB1;
        return 1;
    } else {                      Shift and put 0 into bit 1.
```

```
        *iseed <<= 1;
        return 0;
    }
}
```

Some Primitive Polynomials Modulo 2 (after Watson)

(1, 0)	(51, 6, 3, 1, 0)
(2, 1, 0)	(52, 3, 0)
(3, 1, 0)	(53, 6, 2, 1, 0)
(4, 1, 0)	(54, 6, 5, 4, 3, 2, 0)
(5, 2, 0)	(55, 6, 2, 1, 0)
(6, 1, 0)	(56, 7, 4, 2, 0)
(7, 1, 0)	(57, 5, 3, 2, 0)
(8, 4, 3, 2, 0)	(58, 6, 5, 1, 0)
(9, 4, 0)	(59, 6, 5, 4, 3, 1, 0)
(10, 3, 0)	(60, 1, 0)
(11, 2, 0)	(61, 5, 2, 1, 0)
(12, 6, 4, 1, 0)	(62, 6, 5, 3, 0)
(13, 4, 3, 1, 0)	(63, 1, 0)
(14, 5, 3, 1, 0)	(64, 4, 3, 1, 0)
(15, 1, 0)	(65, 4, 3, 1, 0)
(16, 5, 3, 2, 0)	(66, 8, 6, 5, 3, 2, 0)
(17, 3, 0)	(67, 5, 2, 1, 0)
(18, 5, 2, 1, 0)	(68, 7, 5, 1, 0)
(19, 5, 2, 1, 0)	(69, 6, 5, 2, 0)
(20, 3, 0)	(70, 5, 3, 1, 0)
(21, 2, 0)	(71, 5, 3, 1, 0)
(22, 1, 0)	(72, 6, 4, 3, 2, 1, 0)
(23, 5, 0)	(73, 4, 3, 2, 0)
(24, 4, 3, 1, 0)	(74, 7, 4, 3, 0)
(25, 3, 0)	(75, 6, 3, 1, 0)
(26, 6, 2, 1, 0)	(76, 5, 4, 2, 0)
(27, 5, 2, 1, 0)	(77, 6, 5, 2, 0)
(28, 3, 0)	(78, 7, 2, 1, 0)
(29, 2, 0)	(79, 4, 3, 2, 0)
(30, 6, 4, 1, 0)	(80, 7, 5, 3, 2, 1, 0)
(31, 3, 0)	(81, 4 0)
(32, 7, 5, 3, 2, 1, 0)	(82, 8, 7, 6, 4, 1, 0)
(33, 6, 4, 1, 0)	(83, 7, 4, 2, 0)
(34, 7, 6, 5, 2, 1, 0)	(84, 8, 7, 5, 3, 1, 0)
(35, 2, 0)	(85, 8, 2, 1, 0)
(36, 6, 5, 4, 2, 1, 0)	(86, 6, 5, 2, 0)
(37, 5, 4, 3, 2, 1, 0)	(87, 7, 5, 1, 0)
(38, 6, 5, 1, 0)	(88, 8, 5, 4, 3, 1, 0)
(39, 4, 0)	(89, 6, 5, 3, 0)
(40, 5, 4 3, 0)	(90, 5, 3, 2, 0)
(41, 3, 0)	(91, 7, 6, 5, 3, 2, 0)
(42, 5, 4, 3, 2, 1, 0)	(92, 6, 5, 2, 0)
(43, 6, 4, 3, 0)	(93, 2, 0)
(44, 6, 5, 2, 0)	(94, 6, 5, 1, 0)
(45, 4, 3, 1, 0)	(95, 6, 5, 4, 2, 1, 0)
(46, 8, 5, 3, 2, 1, 0)	(96, 7, 6, 4, 3, 2, 0)
(47, 5, 0)	(97, 6, 0)
(48, 7, 5, 4, 2, 1, 0)	(98, 7, 4, 3, 2, 1, 0)
(49, 6, 5, 4, 0)	(99, 7, 5, 4, 0)
(50, 4, 3, 2, 0)	(100, 8, 7, 2, 0)

A word of caution is: Don't use sequential bits from these routines as the bits of a large, supposedly random, integer, or as the bits in the mantissa of a supposedly random floating-point number. They are not very random for that purpose; see Knuth [1]. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by ± 1 at a rapid "chip rate," so as to spread its spectrum uniformly (but recoverably) across some desired bandpass, or (ii) Monte Carlo exploration of a binary tree, where decisions as to whether to branch left or right are to be made randomly.

Now we do not want you to go through life thinking that there is something special about the primitive polynomial of degree 18 used in the above examples. (We chose 18 because 2^{18} is small enough for you to verify our claims directly by numerical experiment.) The accompanying table [2] lists one primitive polynomial for each degree up to 100. (In fact there exist many such for each degree. For example, see §7.7 for a complete table up to degree 10.)

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 29ff. [1]

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32–9.37.

Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

7.5 Random Sequences Based on Data Encryption

In *Numerical Recipes'* first edition, we described how to use the Data Encryption Standard (DES) [1-3] for the generation of random numbers. Unfortunately, when implemented in software in a high-level language like C, DES is very slow, so excruciatingly slow, in fact, that our previous implementation can be viewed as more mischievous than useful. Here we give a much faster and simpler algorithm which, though it may not be secure in the cryptographic sense, generates about equally good random numbers.

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [4]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly nonlinear bit-mixing function. Figure 7.5.1 shows the flow of information in DES during this mixing. The function g, which takes 32-bits into 32-bits, is called the "cipher function." Meyer and Matyas [4] discuss the importance of the cipher function being nonlinear, as well as other design criteria.

DES constructs its cipher function g from an intricate set of bit permutations and table lookups acting on short sequences of consecutive bits. Apparently, this function was chosen to be particularly strong cryptographically (or conceivably as some critics contend, to have an exquisitely subtle cryptographic flaw!). For our purposes, a different function g that can be rapidly computed in a high-level computer language is preferable. Such a function may weaken the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest g, and smallest number of iterations of the mixing procedure in Figure 7.5.1, such that our output random sequence passes the standard tests that are customarily applied to random number generators. The resulting algorithm will not be DES, but rather a kind of "pseudo-DES," better suited to the purpose at hand.

Following the criterion, mentioned above, that g should be nonlinear, we must give the integer multiply operation a prominent place in g. Because 64-bit registers are not generally accessible in high-level languages, we must confine ourselves to multiplying 16-bit