exhausted. Here is a piece of code for doing both \( G(i) \) and its inverse.

```c
unsigned long igray(unsigned long n, int is)
For zero or positive values of is, return the Gray code of n; if is is negative, return the inverse
Gray code of n
{
  int ish;
  unsigned long ans, idiv;
  if (is >= 0)
    return n ^ (n >> 1);
  ish=1;
  ans=n;
  for (;;)
    { 
      ans ^= (idiv-ans >> ish);
      if (idiv <= 1 || ish == 16) return ans;
      ish <<= 1;
    }
}
```

In numerical work, Gray codes can be useful when you need to do some task that depends intimately on the bits of \( i \), looping over many values of \( i \). Then, if there are economies in repeating the task for values differing by only one bit, it makes sense to do things in Gray code order rather than consecutive order. We saw an example of this in §7.7, for the generation of quasi-random sequences.

CITED REFERENCES AND FURTHER READING:

### 20.3 Cyclic Redundancy and Other Checksums

When you send a sequence of bits from point A to point B, you want to know that it will arrive without error. A common form of insurance is the "parity bit," attached to 7-bit ASCII characters to put them into 8-bit format. The parity bit is chosen so as to make the total number of one-bits (versus zero-bits) either always even ("even parity") or always odd ("odd parity"). Any single bit error in a character will thereby be detected. When errors are sufficiently rare, and do not occur closely bunched in time, use of parity provides sufficient error detection.

Unfortunately, in real situations, a single noise "event" is likely to disrupt more than one bit. Since the parity bit has two possible values (0 and 1), it gives, on average, only a 50% chance of detecting an erroneous character with more than one wrong bit. That probability, 50%, is not nearly good enough for most applications. Most communications protocols use a multibit generalization of the parity bit called a "cyclic redundancy check" or CRC. In typical applications the CRC is 16 bits long (two bytes or two characters), so that the chance of a random error going undetected is 1 in \( 2^{16} = 65536 \). Moreover, 16-bit CRCs have the mathematical
property of detecting all errors that occur in $M$ or fewer consecutive bits, for any length of message. (We prove this below.) Since noise in communication channels tends to be “bursty,” with short sequences of adjacent bits getting corrupted, this consecutive-bit property is highly desirable.

Normally CRCs lie in the province of communications software experts and chip-level hardware designers — people with bits under their fingernails. However, there are at least two kinds of situations where some understanding of CRCs can be useful to the rest of us. First, we sometimes need to be able to communicate with a lower-level piece of hardware or software that expects a valid CRC as part of its input. For example, it can be convenient to have a program generate XMODEM or Kermit [2] packets directly into the communications line rather than having to store the data in a local file.

Second, in the manipulation of large quantities of (e.g., experimental) data, it is useful to be able to tag aggregates of data (whether numbers, records, lines, or whole files) with a statistically unique “key,” its CRC. Aggregates of any size can then be compared for identity by comparing only their short CRC keys. Differing keys imply nonidentical records. Identical keys imply, to high statistical certainty, identical records. If you can’t tolerate the very small probability of being wrong, you can do a full comparison of the records when the keys are identical. When there is a possibility of files or data records being inadvertently or irresponsibly modified (for example, by a computer virus), it is useful to have their prior CRCs stored externally on a physically secure medium, like a floppy disk.

Sometimes CRCs can be used to compress data as it is recorded. If identical data records occur frequently, one can keep sorted in memory the CRCs of previously encountered records. A new record is archived in full if its CRC is different, otherwise only a pointer to a previous record need be archived. In this application one might desire a 4- or 8-byte CRC, to make the odds of mistakenly discarding a different data record be tolerably small; or, if previous records can be randomly accessed, a full comparison can be made to decide whether records with identical CRCs are in fact identical.

Now let us briefly discuss the theory of CRCs. After that, we will give implementations of various (related) CRCs that are used by the official or de facto standard protocols [1-3] listed in the accompanying table.

The mathematics underlying CRCs is “polynomials over the integers modulo 2.” Any binary message can be thought of as a polynomial with coefficients 0 and 1. For example, the message “1100001101” is the polynomial $x^9 + x^5 + x^3 + x^2 + 1$. Since 0 and 1 are the only integers modulo 2, a power of $x$ in the polynomial is either present (1) or absent (0). Polynomials over the integers modulo 2 are a so-called unique factorization domain. This means that any polynomial has a unique factorization into so-called irreducible or “primitive” polynomials — analogs of the prime integers. The polynomial $x^2 + x + 1$ is primitive, while the polynomial $x^2 + 1$ is not: $x^2 + 1 = (x + 1)(x + 1)$. (Remember that the integer arithmetic is done modulo two!) A related theorem says that if a polynomial $p$ divides another polynomial $q$, then it must divide at least one of $q$’s factors.

An $M$-bit long CRC is based on a particular primitive polynomial of degree $M$, called the generator polynomial. The choice of which primitive polynomial to use is only a matter of convention. For 16-bit CRC’s, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the “CCITT polynomial,”
### Conventions and Test Values for Various CRC Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>init</th>
<th>jrev</th>
<th>T</th>
<th>Packet</th>
<th>Format</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMODEM</td>
<td>0</td>
<td>1</td>
<td>1A71</td>
<td>CatMouse987654321</td>
<td>E556</td>
<td>S1S2…S2C1C2 0</td>
</tr>
<tr>
<td>X.25</td>
<td>255</td>
<td>1</td>
<td>1B26</td>
<td>F56E</td>
<td>S1S2…S2C1C2</td>
<td>F0B8</td>
</tr>
<tr>
<td>(no name)</td>
<td>255</td>
<td>1</td>
<td>1B26</td>
<td>F56E</td>
<td>S1S2…S2C1C2 0</td>
<td></td>
</tr>
<tr>
<td>SDLC (IBM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>same as X.25</td>
</tr>
<tr>
<td>HDLC (ISO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>same as X.25</td>
</tr>
<tr>
<td>CRC-CCITT</td>
<td>0</td>
<td>1</td>
<td>14A1</td>
<td>C28D</td>
<td></td>
<td>S1S2…S2C1C2 0</td>
</tr>
<tr>
<td>(no name)</td>
<td>0</td>
<td>1</td>
<td>14A1</td>
<td>C28D</td>
<td>S1S2…S2C1C2</td>
<td>F0B8</td>
</tr>
<tr>
<td>Kermit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>same as CRC-CCITT see Notes</td>
</tr>
</tbody>
</table>

**Notes:** Overbar denotes bit complement. \( S_1 \ldots S_Y \) are character data. \( C_1 \) is CRC’s least significant 8 bits, \( C_2 \) is its most significant 8 bits, so \( CRC = 256C_2 + C_1 \) (shown in hex). Kermit (block check level 3) sends the CRC as 3 printable ASCII characters (sends value +32). These contain, respectively, 4 most significant bits, 6 middle bits, 6 least significant bits.

which is \( x^{16} + x^{12} + x^5 + 1 \). This polynomial is used by all of the protocols listed in the table. Another common choice is the “CRC-16” polynomial \( x^{16} + x^{15} + x^2 + 1 \), which is used for EBCDIC messages in IBM’s BISYNCH[1]. A common 12-bit choice, “CRC-12,” is \( x^{12} + x^{11} + x^3 + x + 1 \). A common 32-bit choice, “AUTODIN-II,” is \( x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 \). For a table of some other primitive polynomials, see §7.4.

Given the generator polynomial \( G \) of degree \( M \) (which can be written either in polynomial form or as a bit-string, e.g., 10001000000100001 for CCITT), here is how you compute the CRC for a sequence of bits \( S \): First, multiply \( S \) by \( x^M \), that is, append \( M \) zero bits to it. Second divide — by long division — \( G \) into \( Sx^M \). Keep in mind that the subtractions in the long division are done modulo 2, so that there are never any “borrows”: Modulo 2 subtraction is the same as logical exclusive-or (XOR). Third, ignore the quotient you get. Fourth, when you eventually get to a remainder, it is the CRC, call it \( C \). \( C \) will be a polynomial of degree \( M - 1 \) or less, otherwise you would not have finished the long division. Therefore, in bit string form, it has \( M \) bits, which may include leading zeros. \( C \) might even be all zeros, see below.) See [3] for a worked example.

If you work through the above steps in an example, you will see that most of what you write down in the long-division tableau is superfluous. You are actually just left-shifting sequential bits of \( S \), from the right, into an \( M \)-bit register. Every time a 1 bit gets shifted off the left end of this register, you zap the register by an XOR with the \( M \) low order bits of \( G \) (that is, all the bits of \( G \) except its leading 1). When a 0 bit is shifted off the left end you don’t zap the register. When the last bit that was originally part of \( S \) gets shifted off the left end of the register, what remains is the CRC.

You can immediately recognize how efficiently this procedure can be implemented in hardware. It requires only a shift register with a few hard-wired XOR taps into it. That is how CRCs are computed in communications devices, by a single
chip (or small part of one). In software, the implementation is not so elegant, since bit-shifting is not generally very efficient. One therefore typically finds (as in our implementation below) table-driven routines that pre-calculate the result of a bunch of shifts and XORs, say for each of 256 possible 8-bit inputs.

We can now see how the CRC gets its ability to detect all errors in $M$ consecutive bits. Suppose two messages, $S$ and $T$, differ only within a frame of $M$ bits. Then their CRCs differ by an amount that is the remainder when $G$ is divided into $(S - T)x^M \equiv D$. Now $D$ has the form of leading zeros (which can be ignored), followed by some 1’s in an $M$-bit frame, followed by trailing zeros (which are just multiplicative factors of $x$). Since factorization is unique, $G$ cannot possibly divide $D$: $G$ is primitive of degree $M$, while $D$ is a power of $x$ times a factor of (at most) degree $M - 1$. Therefore $S$ and $T$ have inevitably different CRCs.

In most protocols, a transmitted block of data consists of some $N$ data bits, directly followed by the $M$ bits of their CRC (or the CRC XORed with a constant, see below). There are two equivalent ways of validating a block at the receiving end. Most obviously, the receiver can compute the CRC of the data bits, and compare it to the transmitted CRC bits. Less obviously, but more elegantly, the receiver can simply compute the CRC of the total block, with $N + M$ bits, and verify that a result of zero is obtained. Proof: The total block is the polynomial $Sx^M + C$ (data left-shifted to make room for the CRC bits). The definition of $C$ is that $Sx^M = QG + C$, where $Q$ is the discarded quotient. But then $Sx^M + C = QG + C + C = QG$ (remember modulo 2), which is a perfect multiple of $G$. It remains a multiple of $G$ when it gets multiplied by an additional $x^M$ on the receiving end, so it has a zero CRC, q.e.d.

A couple of small variations on the basic procedure need to be mentioned. First, when the CRC is computed, the $M$-bit register need not be initialized to zero. Initializing it to some other $M$-bit value (e.g., all 1’s) in effect prefaces all blocks by a phantom message that would have given the initialization value as its remainder. It is advantageous to do this, since the CRC described thus far otherwise cannot detect the addition or removal of any number of initial zero bits. (Loss of an initial bit, or insertion of zero bits, are common “clocking errors.”) Second, one can add (XOR) any $M$-bit constant $K$ to the CRC before it is transmitted. This constant can either be XORed away at the receiving end, or else it just changes the expected CRC of the whole block by a known amount, namely the remainder of dividing $G$ into $Kx^M$. The constant $K$ is frequently “all bits,” changing the CRC into its ones complement. This has the advantage of detecting another kind of error that the CRC would otherwise not find: deletion of an initial 1 bit in the message with spurious insertion of a 1 bit at the end of the block.

The accompanying function icrc implements the above CRC calculation, including the possibility of the mentioned variations. Input to the function is a pointer to an array of characters, and the length of that array. icrc has two “switch” arguments that specify variations in the CRC calculation. A zero or positive value of jinit causes the 16-bit register to have each byte initialized with the value 1. A negative value of jrev causes each input character to be interpreted as its bit-reverse image, and a similar bit reversal to be done on the output CRC. You do not have to understand this; just use the values of jinit and jrev specified in the table. (If you insist on knowing, the explanation is that serial data ports send characters least-significant bit first (!), and many protocols shift bits into the CRC register in exactly the order received.) The table shows how to construct a block
of characters from the input array and output CRC of \texttt{icrc}. You should not need to do any additional bit-reversal outside of \texttt{icrc}.

The switch \texttt{jinit} has one additional use: When negative it causes the input value of the array \texttt{crc} to be used as initialization of the register. If you set \texttt{crc} to the result of the last call to \texttt{icrc}, this in effect appends the current input array to that of the previous call or calls. Use this feature, for example, to build up the CRC of a whole file a line at a time, without keeping the whole file in memory.

The routine \texttt{icrc} is loosely based on the function in [4]. Here is how to understand its operation: First look at the function \texttt{icrc1}. This incorporates one input character into a 16-bit CRC register. The only trick used is that character bits are XORed into the most significant bits, eight at a time, instead of being fed into the least significant bit, one bit at a time, at the time of the register shift. This works because XOR is associative and commutative — we can feed in character bits any time before they will determine whether to zap with the generator polynomial. (The decimal constant 4129 has the generator’s bits in it.)

\begin{verbatim}
unsigned short icrc1(unsigned short crc, unsigned char onech)
Given a remainder up to now, return the new CRC after one character is added. This routine is functionally equivalent to icrc(,,1,1,1), but slower. It is used by icrc to initialize its table.
{
    int i;
    unsigned short ans=(crc ^ onech << 8);
    for (i=0;i<8;i++)
    {
        if (ans & 0x8000)
            ans = (ans << 1) ^ 4129;
        else
            ans <<= 1;
    }
    return ans;
}
\end{verbatim}

Now look at \texttt{icrc}. There are two parts to understand, how it builds a table when it initializes, and how it uses that table later on. Go back to thinking about a character’s bits being shifted into the CRC register from the least significant end. The key observation is that while 8 bits are being shifted into the register’s low end, all the generator zapping is being determined by the bits already in the high end. Since XOR is commutative and associative, all we need is a table of the result of all this zapping, for each of 256 possible high-bit configurations. Then we can play catch-up and XOR an input character into the result of a lookup into this table. The only other content to \texttt{icrc} is the construction at initialization time of an 8-bit bit-reverse table from the 4-bit table stored in \texttt{it}, and the logic associated with doing the bit reversals. References [4-6] give further details on table-driven CRC computations.

\begin{verbatim}
typedef unsigned char uchar;
#define LOBYTE(x) ((uchar)((x) & 0xFF))
#define HIBYTE(x) ((uchar)((x) >> 8))

unsigned short icrc(unsigned short crc, unsigned char *bufptr,
                        unsigned long len, short jinit, int jrev)
Computes a 16-bit Cyclic Redundancy Check for an array bufptr of length len bytes, using any of several conventions as determined by the settings of jinit and jrev (see accompanying
Here: Call it once with nonzero bits only in its least significant byte (which makes for some simplification).

Wagner and the check digit to numbers that are handled by human beings (e.g., typed into a computer). Check digits need to be proof against the kinds of highly structured errors that humans tend to make, such as transposing consecutive digits. Wagner and Putter give an interesting introduction to this subject, including specific algorithms.

What if you need a 32-bit checksum? For a true 32-bit CRC, you will need a generating polynomial that is primitive modulo 2, and has nonleading, nonzero bits only in its least significant byte (which makes for some simplification). The idea of table lookup on only the most significant byte of the CRC register goes through unchanged.

If you do not care about the M-consecutive bit property of the checksum, but rather only need a statistically random 32 bits, then you can use icrc as given here: Call it once with jrev = 1 to get 16 bits, and again with jrev = -1 to get another 16 bits. The internal bit reversals make these two 16-bit CRCs in effect totally independent of each other.

Other Kinds of Checksums

Quite different from CRCs are the various techniques used to append a decimal “check digit” to numbers that are handled by human beings (e.g., typed into a computer). Check digits need to be proof against the kinds of highly structured errors that humans tend to make, such as transposing consecutive digits. Wagner and Putter give an interesting introduction to this subject, including specific algorithms.

Checksums now in widespread use vary from fair to poor. The 10-digit ISBN (International Standard Book Number) that you find on most books, including this one, uses the check equation

\[ 10d_1 + 9d_2 + 8d_3 + \cdots + 2d_9 + d_{10} = 0 \pmod{11} \]  

where \( d_{10} \) is the right-hand check digit. The character “X” is used to represent a check digit value of 10. Another popular scheme is the so-called “IBM check,” often
used for account numbers (including, e.g., MasterCard). Here, the check equation is
\[ 2 \#d_1 + d_2 + 2 \#d_3 + d_4 + \cdots = 0 \pmod{10} \quad (20.3.2) \]
where \(2 \# d\) means, “multiply \(d\) by two and add the resulting decimal digits.” United
States banks code checks with a 9-digit processing number whose check equation is
\[ 3a_1 + 7a_2 + a_3 + 3a_4 + 7a_5 + a_6 + 3a_7 + 7a_8 + a_9 = 0 \pmod{10} \quad (20.3.3) \]
The bar code put on many envelopes by the U.S. Postal Service is decoded by
removing the single tall marker bars at each end, and breaking the remaining bars
into 6 or 10 groups of five. In each group the five bars signify (from left to right)
the integers modulo 10. The check equation is
\[ \sum d_i = 0 \pmod{10} \quad (20.3.4) \]
None of these schemes is close to optimal. An elegant scheme due to Verhoeff
is described in [7]. The underlying idea is to use the ten-element dihedral group \(D_5\),
which corresponds to the symmetries of a pentagon, instead of the cyclic group of
the integers modulo 10. The check equation is
\[ a_1 \cdot f(a_2) \cdot f^2(a_3) \cdot \cdots \cdot f^{n-1}(a_n) = 0 \quad (20.3.5) \]
where \(\cdot\) is (noncommutative) multiplication in \(D_5\), and \(f^i\) denotes the \(i\)th iteration
of a certain fixed permutation. Verhoeff’s method finds all single errors in a string,
and all adjacent transpositions. It also finds about 95% of twin errors (\(aa \rightarrow bb\)),
jump transpositions (\(acb \rightarrow bca\)), and jump twin errors (\(aca \rightarrow bcb\)). Here is an implementation:

```c
int decchk(char string[], int n, char *ch)
{ // Decimal check digit computation or verification. Returns as ch a check digit for appending to string[1..n] that is, for storing into string[n+1]. In this mode, ignore the returned boolean (integer) value. If string[1..n] already ends with a check digit (string[n]), returns the function value true (1) if the check digit is valid, otherwise false (0). In this mode, ignore the returned value of ch. Note that string and ch contain ASCII characters corresponding to the digits 0-9, not byte values in that range. Other ASCII characters are allowed in string, and are ignored in calculating the check digit.
    //
    int j,k=0,m=0;
    static int ip[10][8]={
      0,1,5,8,9,4,2,7,1,5, 8,9,4,2,7,0,2,7,0,1,
      8,9,4,2,7,0,1,5,8,9, 5,8,9,4,2,7,0,1,6,3,
      6,3,6,3,6,3,7,0,1,5, 8,9,4,2,8,9,4,2,7,0,1,5,9,4,2,7,0,1,5,8,
      2,3,4,0,1,7,8,9,5,6, 3,4,0,1,2,8,9,5,6,7, 4,0,1,2,3,9,5,6,7,8,
      8,9,8,7,6,0,4,3,2,1, 6,5,9,8,7,1,0,4,3,2, 7,6,5,9,8,2,1,0,4,3,
      8,7,6,5,9,3,2,1,0,4, 9,8,7,6,5,4,3,2,1,0};
    static int ij[10][10]={
      0,1,2,3,4,5,6,7,8,9, 1,2,3,4,5,6,7,8,9,0,
      2,3,4,0,1,7,8,9,5,6, 3,4,0,1,2,8,9,5,6,7, 4,0,1,2,3,9,5,6,7,8,
      8,9,8,7,6,0,4,3,2,1, 6,5,9,8,7,1,0,4,3,2, 7,6,5,9,8,2,1,0,4,3,
      8,7,6,5,9,3,2,1,0,4, 9,8,7,6,5,4,3,2,1,0};
    // Group multiplication and permutation tables.

    for (j=0; j<8; j++) // Look at successive characters.
        c=string[j];
    if (c >= 48 && c <= 57) // Ignore everything except digits.
        m=ip[c-48][j];
    }
```
20.4 Huffman Coding and Compression of Data

A lossless data compression algorithm takes a string of symbols (typically ASCII characters or bytes) and translates it reversibly into another string, one that is on the average of shorter length. The words “on the average” are crucial; it is obvious that no reversible algorithm can make all strings shorter — there just aren’t enough short strings to be in one-to-one correspondence with longer strings. Compression algorithms are possible only when, on the input side, some strings, or some input symbols, are more common than others. These can then be encoded in fewer bits than rarer input strings or symbols, giving a net average gain.

There exist many, quite different, compression techniques, corresponding to different ways of detecting and using departures from equiprobability in input strings. In this section and the next we shall consider only variable length codes with defined word inputs. In these, the input is sliced into fixed units, for example ASCII characters, while the corresponding output comes in chunks of variable size. The simplest such method is Huffman coding [1], discussed in this section. Another example, arithmetic compression, is discussed in §20.5.

At the opposite extreme from defined-word, variable length codes are schemes that divide up the input into units of variable length (words or phrases of English text, for example) and then transmit these, often with a fixed-length output code. The most widely used code of this type is the Ziv-Lempel code [2]. References [3-6] give the flavor of some other compression techniques, with references to the large literature.

The idea behind Huffman coding is simply to use shorter bit patterns for more common characters. We can make this idea quantitative by considering the concept of entropy. Suppose the input alphabet has $N_{ch}$ characters, and that these occur in the input string with respective probabilities $p_i$, $i = 1, \ldots, N_{ch}$, so that $\sum p_i = 1$. Then the fundamental theorem of information theory says that strings consisting of