High-Speed Computation of Cyclic Redundancy Checks

Eric E. Johnson

November 1995

NMSU-ECE-95-011

High-Speed Computation of Cyclic Redundancy Checks

Eric E. Johnson
Klipsch School of Electrical and Computer Engineering
New Mexico State University

November 1995

Abstract

Cyclic redundancy checks (CRCs) are widely used in error-prone channels to detect corruption of data blocks. This report describes implementations of both the standard bit-by-bit technique for computing CRCs and a high-speed technique that employs precomputed tables to compute CRCs on a byte-by-byte basis. The technique could be further extended to compute CRCs word-by-word.

INTRODUCTION

When data are stored on or communicated through media that may introduce errors, some form of error detection or error detection and correction coding is usually employed. When error rates are sufficiently low that detection alone is acceptable, cyclic redundancy check (CRC) codes are often used.

Mathematically, a CRC is computed for a block of data by treating that data block as a string of binary coefficients of a polynomial, which is divided by a "generator polynomial," with the remainder of this division used as the CRC. Consider a string of N data bits, $i_{N-1} \ldots i_0$, in which i_{N-1} is the first bit sent. We define the *data polynomial* to be[1]

$$i(x) = x^r (i_{N-1} x^{N-1} + i_{N-2} x^{N-2} + \ldots + i_1 x + i_0).$$

The multiplication by x^r shifts the data polynomial r bits to the left to make room for an r-bit CRC o(x), which is the remainder when the data polynomial d(x) is divided by the *generator polynomial*

$$g(x) = x^r + g_{r-1} x^{r-1} + g_{r-2} x^{r-2} + \ldots + g_1 x + g_0.$$

That is, $i(x) = h(x)\, g(x) + o(x)$ for some h(x). Note that o(x), as the remainder of a division by g(x), is necessarily of order r–1 or less, and therefore can be represented by a string of r bits.

The CRC is appended to the string of data bits to form a string of N+r bits $c(x) = i(x) + o(x)$

$$c(x) = x^r (i_{N-1} x^{N-1} + i_{N-2} x^{N-2} + \ldots + i_1 x + i_0) + o_{r-1} x^{r-1} + o_{r-2} x^{r-2} + \ldots + o_1 x + o_0.$$

[1] All additions are modulo-2.

When this bit string is sent through a channel (e.g., stored on a disk and subsequently retrieved, or received from a communications channel), the presence of errors may be detected by recomputing the CRC using the received data bits î and verifying that the newly computed CRC matches that received at the end of the bit string (ô). If the computed CRC does not match ô, one or more errors have been introduced by the channel. If the computed CRC matches ô, the data block is assumed to be error free, although there is a small probability that undetected errors have occurred. For a suitably chosen g(x), the probability of undetected errors is approximately 2^{-r}.

It may be more convenient to instead simply divide the polynomial corresponding to the entire received bit stream (î + ô) by the generator polynomial. This avoids the need to know *a priori* the length of the data block.

Because i(x) = h(x) g(x) + o(x), we have i(x) + o(x) = h(x) g(x) modulo 2. Thus, if the bit stream sent is divided by g(x), the remainder will be 0, because i(x) + o(x) is an even multiple of g(x). Likewise, if the received bit stream î(x) + ô(x) is divided by g(x) and a remainder of 0 results, the received block is probably identical to the bits sent, and therefore error-free.

SHIFT REGISTER CRC COMPUTATION

A standard technique for computing CRCs employs an r-bit feedback shift register as shown in Figure 1 to perform the polynomial division [1].

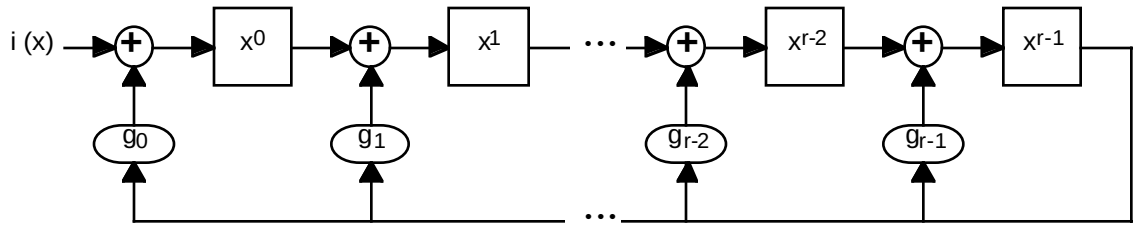

Figure 1: Shift register computation of CRC *(after Rao and Fujiwara)*

This circuit operates in a fashion similar to manual long division. The storage elements in Figure 1 hold the coefficients of the divisor corresponding to the indicated powers of x. For each cycle of the division algorithm, the x^{r-1} coefficient at the end of one cycle will become the x^r coefficient for the next cycle. If this x^r coefficient is 1, the generator polynomial will be subtracted from the divisor. Because the x^r coefficient of the generator polynomial is always 1, the resulting x^r coefficient in the divisor will always be 0, and need not be stored.

Because the generator polynomial is of order r, only the r next-most-significant bits can be affected by the subtraction, so only r storage elements are needed. After each subtraction (actually modulo-2 addition), the resulting modified coefficients of the divisor are stored in the shift register.

The data bits are fed into the circuit in the order i_{N-1} ... i_0. Because i(x) is scaled by x^r, i_0 must be followed by r zero bits. After i(x) has been divided by g(x), the contents of the shift register are the remainder o(x), which can then be shifted out. The most-significant bit o_{r-1} (found in the x^{r-1} element of the shift register) is sent first, followed in order by the remaining bits.

Algorithm for Bit-by-Bit CRC Computation

Although the shift register approach to computing CRCs is usually implemented in hardware, when bit-by-bit processing efficiency in software is adequate, the following algorithm can be used:

- Let R represent the contents of the r-bit shift register, with the most significant bit denoted R_{r-1} and so on.
- Let G represent the r least-significant coefficients of the generator polynomial, and let Gr1 = G right-shifted one bit position.
- Let i represent the input bit for each cycle of the computation.

The computation of the CRC proceeds as follows, using C notation in which ^ represent the exclusive-or (modulo-2 addition) operation and << n represents left shifting by n bits:

1. Initialize R to 0.
2. For each input bit i

   if (R_{r-1} == 1)

   R = ((R ^ Gr1) << 1) + (i ^ 1)

   otherwise

   R = (R << 1) + i

Shift Register Computation of CRC-CCITT

To illustrate the process involved, this section presents a program that computes the CRC used in X.25, also known as CRC-CCITT:

- This is a 16-bit CRC, so r = 16.
- The generator polynomial $g(x) = x^{16} + x^{12} + x^5 + 1$.
- G = 0x1021 and Gr1 = 0x0810.

The C program on the following two pages contains the following functions:

- `ResetCRC()`, which must be called before each new data block. It simply resets the register `R` to 0.
- `UpdateCRC`(x), which processes the 8 bits of its byte-size argument `x`. The `printf()` statements provide a running commentary on the computation, but they can, of course, be deleted when this is not desired.
- `CheckCRC(fp, length)` resets the shift register and then calls `UpdateCRC` for each of `length` bytes read from the file attached to file pointer `fp`.

The main program reads a file from `stdin`, copying it to a file `tmp`, and computing the CRC, which is appended to `tmp` after the end of the input file is encountered. It then calls `CheckCRC` with a pointer to the `tmp` file and prints the results of checking the CRC.

```
/*  srtst.c    eej    11/25/95
*
*   implements CRC-CCITT using shift register
*/

#include <stdio.h>

static unsigned int R, Gr1 = 0x0810;

void
ResetCRC()
{
  R = 0;
}

void
UpdateCRC(char x)
{
    int i, k;

printf("\nUpdateCRC(%02x)\n", x);
    for (k=0; k<8; k++) {
      i = (x >> 7) & 1;  /* msb */
printf("  %04x < %1x  ->  ", R, i);
      if (R & 0x8000)  /* is msb of R == 1? */
        R = ((R ^ Gr1) << 1) + (i ^ 1);
      else
        R = (R << 1) + i;
      R &= 0x0ffff;
printf("%04x\n", R);
      x <<= 1;
    }
}

/***********************************
*            CheckCRC              *
*                                  *
*  Computes CRC over buffer of     *
*  given length.  Returns 0 if     *
*  last 2 bytes were correct CRC   *
***********************************/

long
CheckCRC(FILE *fp, int length)
{
    int i;

    ResetCRC();
    for (i = 0; i < length; i++) {
        UpdateCRC(getc(fp));
    }
    return R;
}
```

```
main()
{
  FILE *fp;
  int c, length = 0;
  long crcSent;

  if ((fp = fopen("tmp", "w")) == NULL) {
    printf("Can't open tmp file.  Exiting\n");
    exit(-1);
  }
  ResetCRC();
  while ((c = getchar()) != EOF) {
    UpdateCRC(c);
    putc(c, fp);
    length++;
  }
  UpdateCRC(0); length++;
  UpdateCRC(0); length++;
  printf("CRC is %04x\n", R);
  crcSent = R;
  putc(((R >> 8) & 0xff), fp);
  putc((R & 0xff), fp);
  fclose(fp);

  if ((fp = fopen("tmp", "r")) == NULL) {
    printf("Can't read tmp file.  Exiting\n");
    exit(-2);
  }
  if (CheckCRC(fp, length) == 0)
    printf("CRC checks.\n");
  else
    printf("Computed CRC doesn't match\n");
}
```

TABLE-DRIVEN CRC COMPUTATION

A more efficient approach to CRC computation in software was described by Sarwate [2]. This technique uses a table of precomputed effects on the shift register of 8-bit bytes, which allows the computation to run at one cycle per byte (instead of one cycle per bit).

Precomputation

The table used by this algorithm contains words of length r that represent the composite effect of processing an entire 8-bit byte through the polynomial division algorithm described previously. The table is indexed by the contents of the byte to be processed, and therefore contains $2^8 = 256$ entries. The entries are computed as follows:

1. First, compute 8 basis polynomials $g_j(x)$, which represent the effects of input bits that will enter the shift register 7 - j bit times in the future.
2. Then, compute the table entries by adding (modulo-2) the basis polynomials corresponding to 1 bits in the index for each entry.

This process is illustrated in the following code fragments. The CRC-CCITT is again used in this example. Note once again that only r bits need be stored for the basis polynomials and the table entries.

```
/* compute the basis polynomials */
/* g[0] = 0b0001 0000 0010 0001; */
g[0] = 0x1021;
for (j=1; j<8; j++)  {
  g[j] = g[j-1] << 1;
  if ((g[j-1] >> 15) & 1)
    g[j] ^= g[0];
    g[j] &= 0x0ffff;
}

/* compute the table entries */
for (i=0; i<256; i++)  {
  f = 0;
  T = i;
  for (j=0; j<8; j++)  {
    if (T & 1) f ^= g[j];
    T >>= 1;
  }
  /* f now contains table entry i */
}
```

CRC Computation

The precomputed table can either be generated each time the program is started, or written to an auxiliary file that is read when the CRC computation program is compiled (`ccitttab.h` in the following example program). CRC computation uses a shift register of r bits (`crcsum` in the example), into which input bytes and table entries are summed and shifted in `UpdateCRC()`:

```
/*  crctst.c    eej    11/26/95
*
*   implements CRC-CCITT using table lookup
*/

#include <stdio.h>
#include "ccitttab.h"

static long crcSum;
static char *nextByte;
static int restart;

void
ResetCRC()
{
    crcSum = 0;
}

void
UpdateCRC(char x)
{
    int tmp;

printf("UpdateCRC(%02x)  %04x -> ", x, crcSum);
    tmp = ((x ^ (crcSum >> 8)) & 0xff);
    crcSum = ((crcSum << 8) ^ CrcTab[tmp]) & 0xffff;
printf("%04x\n", crcSum);
}

/************************************
*                                   *
*            CheckCRC               *
*                                   *
*  Computes CRC over buffer of      *
*  given length.  Returns 0 if      *
*  last 2 bytes were correct CRC    *
*                                   *
************************************/

long
CheckCRC(FILE *fp, int length)
{
    int i;
    long tmp;

    ResetCRC();
    for (i = 0; i < length; i++) {
        UpdateCRC(getc(fp));
    }
    return crcSum;
}
```

```
main()
{
  FILE *fp;
  int c, length = 0;
  long crcSent;

  if ((fp = fopen("crctmp", "w")) == NULL) {
    printf("Can't open crctmp file.  Exiting\n");
    exit(-1);
  }
  ResetCRC();
  while ((c = getchar()) != EOF) {
    UpdateCRC(c);
    putc(c, fp);
    length++;
  }
  /* UpdateCRC(0); length++;
     UpdateCRC(0); length++; */
  length += 2;
  printf("CRC is %04x\n", crcSum);
  crcSent = crcSum;
  putc(((crcSum >> 8) & 0xff), fp);
  putc((crcSum & 0xff), fp);
  fclose(fp);

  if ((fp = fopen("crctmp", "r")) == NULL) {
    printf("Can't read crctmp file.  Exiting\n");
    exit(-2);
  }
  if (CheckCRC(fp, length) == 0)
    printf("CRC checks.\n");
  else
    printf("Computed CRC doesn't match\n");
}
```

Computation of ADCCP CRC

In a departure from usual CRC practice, the ADCCP CRC [3] specifies that the shift register shall be initialized to all 1's before computing the CRC (the first 2 input bytes may instead be inverted), and that the remainder be inverted before sending it over the channel.

The table-driven algorithm for computing the ADCCP CRC is modified from the example given above by not only initializing R to 0x0ffff and inverting the remainder before it is appended to the data, but also by inverting the received CRC in the CRC checking function before processing those 2 CRC bytes.

This is illustrated in the following example program, which may be used to support the CRC function of the high-frequency radio automatic link establishment standards [4, 5].

```
/*  crctst.c    eej    11/26/95
*
*   implements CRC-CCITT using table lookup
*   ADCCP version 1: initialize crcSum (C1C0) to 1's
*/

#include <stdio.h>
#include "ccitttab.h"

static long crcSum;
static char *nextByte;

void
ResetCRC()
{
  crcSum = 0x0ffff;
}

void
UpdateCRC(char x)
{
    int tmp;

printf("UpdateCRC(%02x)  %04x -> ", x, crcSum);
    tmp = ((x ^ (crcSum >> 8)) & 0xff);
    crcSum = ((crcSum << 8) ^ CrcTab[tmp]) & 0xffff;
printf("%04x\n", crcSum);
}

/************************************
*                                   *
*            CheckCRC               *
*                                   *
*  Computes CRC over buffer of      *
*  given length.  Returns 0 if      *
*  last 2 bytes were correct CRC    *
*                                   *
************************************/

long
CheckCRC(FILE *fp, int length)
{
  int i;
  long tmp;

  ResetCRC();
  for (i = 0; i < length-2; i++) {
   UpdateCRC(getc(fp));
  }
  for (; i < length; i++) {
   UpdateCRC(getc(fp) ^ 0x0ff);
  }
  return crcSum;
}
```

```
main()
{
  FILE *fp;
  int c, length = 0;
  long crcSent;

  if ((fp = fopen("crctmp", "w")) == NULL) {
    printf("Can't open crctmp file.  Exiting\n");
    exit(-1);
  }
  ResetCRC();
  while ((c = getchar()) != EOF) {
    UpdateCRC(c);
    putc(c, fp);
    length++;
  }
  length += 2;
  crcSum ^= 0x0ffff;
  printf("CRC is %04x\n", crcSum);
  crcSent = crcSum;
  putc(((crcSum >> 8) & 0xff), fp);
  putc((crcSum & 0xff), fp);
  fclose(fp);

  if ((fp = fopen("crctmp", "r")) == NULL) {
    printf("Can't read crctmp file.  Exiting\n");
    exit(-2);
  }
  if (CheckCRC(fp, length) == 0)
    printf("CRC checks.\n");
  else
    printf("Computed CRC doesn't match\n");
}
```

```
/*  the file ccitttab.h contains the precomputed table  */

int CrcTab[256] =
  {0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
   0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
   0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
   0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
   0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
   0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
   0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
   0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
   0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
   0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
   0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
   0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
   0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
   0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
   0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
   0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
   0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
   0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
   0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
   0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
   0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
   0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
   0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
   0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
   0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
   0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
   0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
   0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
   0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
   0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
   0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
   0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0};
```

Computation of 32-bit CRC

The table-driven technique can also be used to speed the computation of 32-bit CRCs. Such a CRC is specified for use with the high-frequency radio data link protocol (HFDLP) [6]:

$$g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The code on the folowing pages presents an implementation of this 32-bit CRC scheme (which uses the ADCCP variation).

```
#include "crctable.h"

static long crcSum;

void
ResetCRC()
{
    crcSum = 0xffffffff;
}

void
UpdateCRC(char x)
{
    int tmp;

    tmp = ((x ^ (crcSum >> 24)) & 0xff);
    crcSum = (crcSum << 8) ^ CrcTab[tmp];
}

void
AppendByte(char x)
{
    UpdateCRC(x & 0xff);
    *nextByte++ = x & 0xff;
}



/***********************************
*                                  *
*            CheckCRC              *
*                                  *
*  Computes CRC over buffer of     *
*  given length.  Returns 0 if     *
*  last 4 bytes were correct CRC   *
*                                  *
***********************************/

long
CheckCRC(char *buffer, int length)
{
  int i;
  long tmp;

  ResetCRC();
  for (i = 0; i < length-4; i++) {
   UpdateCRC(*buffer++);
  }
  for (; i < length; i++) {
   UpdateCRC((*buffer++) ^ 0x0ff);
  }
  return crcSum;
}
```

```
/* crctable.h    eej    2/16/95
 *
 * table for computation of CRC on a byte basis */

long CrcTab[256] = {0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9,
                    0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,
                    0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,
                    0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,
                    0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9,
                    0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,
                    0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011,
                    0x791d4014, 0x7ddc5da3, 0x709f7b7a, 0x745e66cd,
                    0x9823b6e0, 0x9ce2ab57, 0x91a18d8e, 0x95609039,
                    0x8b27c03c, 0x8fe6dd8b, 0x82a5fb52, 0x8664e6e5,
                    0xbe2b5b58, 0xbaea46ef, 0xb7a96036, 0xb3687d81,
                    0xad2f2d84, 0xa9ee3033, 0xa4ad16ea, 0xa06c0b5d,
                    0xd4326d90, 0xd0f37027, 0xddb056fe, 0xd9714b49,
                    0xc7361b4c, 0xc3f706fb, 0xceb42022, 0xca753d95,
                    0xf23a8028, 0xf6fb9d9f, 0xfbb8bb46, 0xff79a6f1,
                    0xe13ef6f4, 0xe5ffeb43, 0xe8bccd9a, 0xec7dd02d,
                    0x34867077, 0x30476dc0, 0x3d044b19, 0x39c556ae,
                    0x278206ab, 0x23431b1c, 0x2e003dc5, 0x2ac12072,
                    0x128e9dcf, 0x164f8078, 0x1b0ca6a1, 0x1fcdbb16,
                    0x018aeb13, 0x054bf6a4, 0x0808d07d, 0x0cc9cdca,
                    0x7897ab07, 0x7c56b6b0, 0x71159069, 0x75d48dde,
                    0x6b93dddb, 0x6f52c06c, 0x6211e6b5, 0x66d0fb02,
                    0x5e9f46bf, 0x5a5e5b08, 0x571d7dd1, 0x53dc6066,
                    0x4d9b3063, 0x495a2dd4, 0x44190b0d, 0x40d816ba,
                    0xaca5c697, 0xa864db20, 0xa527fdf9, 0xa1e6e04e,
                    0xbfa1b04b, 0xbb60adfc, 0xb6238b25, 0xb2e29692,
                    0x8aad2b2f, 0x8e6c3698, 0x832f1041, 0x87ee0df6,
                    0x99a95df3, 0x9d684044, 0x902b669d, 0x94ea7b2a,
                    0xe0b41de7, 0xe4750050, 0xe9362689, 0xedf73b3e,
                    0xf3b06b3b, 0xf771768c, 0xfa325055, 0xfef34de2,
                    0xc6bcf05f, 0xc27dede8, 0xcf3ecb31, 0xcbffd686,
                    0xd5b88683, 0xd1799b34, 0xdc3abded, 0xd8fba05a,
                    0x690ce0ee, 0x6dcdfd59, 0x608edb80, 0x644fc637,
                    0x7a089632, 0x7ec98b85, 0x738aad5c, 0x774bb0eb,
                    0x4f040d56, 0x4bc510e1, 0x46863638, 0x42472b8f,
                    0x5c007b8a, 0x58c1663d, 0x558240e4, 0x51435d53,
                    0x251d3b9e, 0x21dc2629, 0x2c9f00f0, 0x285e1d47,
                    0x36194d42, 0x32d850f5, 0x3f9b762c, 0x3b5a6b9b,
                    0x0315d626, 0x07d4cb91, 0x0a97ed48, 0x0e56f0ff,
                    0x1011a0fa, 0x14d0bd4d, 0x19939b94, 0x1d528623,
                    0xf12f560e, 0xf5ee4bb9, 0xf8ad6d60, 0xfc6c70d7,
                    0xe22b20d2, 0xe6ea3d65, 0xeba91bbc, 0xef68060b,
                    0xd727bbb6, 0xd3e6a601, 0xdea580d8, 0xda649d6f,
                    0xc423cd6a, 0xc0e2d0dd, 0xcda1f604, 0xc960ebb3,
                    0xbd3e8d7e, 0xb9ff90c9, 0xb4bcb610, 0xb07daba7,
                    0xae3afba2, 0xaafbe615, 0xa7b8c0cc, 0xa379dd7b,
                    0x9b3660c6, 0x9ff77d71, 0x92b45ba8, 0x9675461f,
                    0x8832161a, 0x8cf30bad, 0x81b02d74, 0x857130c3,
                    0x5d8a9099, 0x594b8d2e, 0x5408abf7, 0x50c9b640,
                    0x4e8ee645, 0x4a4ffbf2, 0x470cdd2b, 0x43cdc09c,
```

```
0x7b827d21, 0x7f436096, 0x7200464f, 0x76c15bf8,
0x68860bfd, 0x6c47164a, 0x61043093, 0x65c52d24,
0x119b4be9, 0x155a565e, 0x18197087, 0x1cd86d30,
0x029f3d35, 0x065e2082, 0x0b1d065b, 0x0fdc1bec,
0x3793a651, 0x3352bbe6, 0x3e119d3f, 0x3ad08088,
0x2497d08d, 0x2056cd3a, 0x2d15ebe3, 0x29d4f654,
0xc5a92679, 0xc1683bce, 0xcc2b1d17, 0xc8ea00a0,
0xd6ad50a5, 0xd26c4d12, 0xdf2f6bcb, 0xdbee767c,
0xe3a1cbc1, 0xe760d676, 0xea23f0af, 0xeee2ed18,
0xf0a5bd1d, 0xf464a0aa, 0xf9278673, 0xfde69bc4,
0x89b8fd09, 0x8d79e0be, 0x803ac667, 0x84fbdbd0,
0x9abc8bd5, 0x9e7d9662, 0x933eb0bb, 0x97ffad0c,
0xafb010b1, 0xab710d06, 0xa6322bdf, 0xa2f33668,
0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4};
```

REFERENCES

1. T.R.N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice-Hall, 1989.

2. D.V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up," *Communications of the ACM* **31** (8), August 1988, pp. 1008-1013.

3. American National Standard X3.66-1979, *Advanced Data Communication Control Procedures*.

4. MIL-STD-188-141A, Notice 2, *Interoperability and Performance Standards for Medium and High-Frequency Radio Equipment*.

5. FED-STD-1045A, *Telecommunications: HF Radio Automatic Link Establishment*.

6. FED-STD-1052, Appendix B, *HF Data Link Protocol*.