Implementing The CCITT Cyclical Redundancy Check

By Bob Felice, June 17, 2007

Post a Comment

An error detection scheme that does not impose any additional transmission overhead

The CCITT-CRC error detection scheme was first employed (with some minor modifications) by IBM in its SDLC data link protocol and is used today in other modern data link protocols such as HDLC, SS7, and ISDN. Like a checksum, the CCITT-CRC does not impose any additional transmission overhead at the character level. It can detect errors in any arbitrary number of bits of data, and its error detection rate is 99.9984 percent, worse case.

Some rather powerful math stands behind the CCITT-CRC. Fortunately, the reader doesn't need to understand the math in order to use the algorithm. The basic idea is to treat the entire message as a (rather long) binary number, which both the sender and receiver divide using the same divisor. The quotient is discarded, and the remainder is sent as the CRC. If the message is received without error, the receiver's calculation will match the sender's calculation, and the CRC's will agree.

(This is a gross simplification of the process. The CRC is actually the one's complement of the remainder obtained from the modulo 2 division of the message by a generation polynomial. The CCITT-CRC uses:

```
x<sup>16</sup> + x<sup>12</sup> + x<sup>5</
```

for the generator polynomial. The generator is part of the standard established by the CCITT, an international standards body that publishes recommendations dealing with telephony and data communications.)

The elegance of this approach is that the division, which looks as though it ought to be a complicated process, can be implemented in hardware using a shift register and a few exclusive-OR gates.

The division can also be implemented in software, as the function **crc16** (Listing One) demonstrates.

```
/*****************************************
//
// crc16.c - generate a ccitt 16 bit cycli
//
//      The code in this module generates
//
*******************************************

/*
//                               16  12
// The CCITT CRC 16 polynomial is X + X +
// In binary, this is the bit pattern 1 00
//  is 0x11021.
// A 17 bit register is simulated by testi
//  the data, which affords us the luxury
//  16 bit value, 0x1021.
// Due to the way in which we process the
//  are stored in reverse order. This make
*/
#define POLY 0x8408

/*
// note: when the crc is included in the m
//      0xF0B8, before the compliment and
//      0x0F47, after compliment, before t
//      0x470F, after the compliment and t
*/

extern  crc_ok;
int     crc_ok = 0x470F;

/*******************************************
//
// crc16() - generate a 16 bit crc
```

Recent Articles

Dr. Dobb's Archive
Farewell, *Dr. Dobb's*
Jolt Awards 2015: Coding Tools
Thriving Among the APIs
The Long Death of Project Hosting Sites

Most Popular

Stories Blogs

RESTful Web Services: A Tutorial
Lambda Expressions in Java 8
Developer Reading List: The Must-Have Books for JavaScript
An Algorithm for Compressing Space and Time
Why Build Your Java Projects with Gradle Rather than Ant or Maven?

Upcoming Events

Live Events WebCasts

INsecurity Conference - An Event Like No Other - INsecurity
Black Hat Trainings - October 22 & 23 - INsecurity
Are You Looking to Learn About New Cybersecurity Threats & Challenges? Then INsecurity Is For You! - INsecurity

Featured Reports

What's this?

Return of the Silos
Strategy: The Hybrid Enterprise Data Center
Research: State of the IT Service Desk
Will IPv6 Make Us Unsafe?
Database Defenses

More >>

Featured Whitepapers

What's this?

Top Six Things to Consider with an Identity as a Service Solution
Market Overview: Vulnerability Management
FAQ: Cloud Operations Platform for AWS
Red Hat cloud a road map to government cloud computing based on openness, portability, and choice
Challenging Some of the Myths About Static Code Analysis

More >>

Most Recent Premium Content

Digital Issues

```
//
//
// PURPOSE
//       This routine generates the 16 bit 
//       data using the ccitt polynomial ge
//
// CALLING SEQUENCE
//       crc = crc16(data, len);
//
// PARAMETERS
//       data    <-- address of start of da
//       len     <-- length of data block
//
// RETURNED VALUE
//       crc16 value. data is calcuated usi
//
// NOTES
//       The CRC is preset to all 1's to de
//         of leading zero's.
//       The CRC (a 16 bit value) is genera
//       Two ways to verify the integrity o
//         or block of data:
//         1) Calculate the crc on the data
//            calculated previously. The lo
//            known.
/          2) Append the calculated crc to 
//            the crc of the data and its c
//            value in "crc_ok", the data i
//
// PSEUDO CODE:
//       initialize crc (-1)
//       DO WHILE count NE zero
//         DO FOR each bit in the data byte
//           IF (LSB of crc) EOR (LSB of da
//              crc := (crc / 2) EOR polynom
//           ELSE
//              crc := (crc / 2)
//           FI
//         OD
//       OD
//       1's compliment and swap bytes in c
//       RETURN crc
//
******************************************
unsigned short crc16(data_p, length)
char *data_p;
unsigned short length;
{
       unsigned char i;
       unsigned int data;
       unsigned int crc;

       crc = 0xffff;

       if (length == 0)
              return (~crc);

       do {
              for (i = 0 data = (unsigned 
                  i < 8;
                  i++, data >>= 1) {
                    if ((crc & 0x0001) ^ (
                           crc = (crc >> 1
                    else
                           crc >>= 1;
              }
       } while (--length);

       crc = ~crc;

       data = crc;
       crc = (crc << 8) | (data >> 8 & 0xF

       return (crc);
}
```

Listing One.

The processor overhead involved in calculating the checksum is not too bad when you consider the large number of errors that the algorithm can detect. Two noteworthy items: the CCITT-CRC is calculated on the data bits in the order that they are sent (least significant bit first); and the 16-bit CRC is itself sent least significant byte first.

The initial value of the CRC, known as the "preset," can be either 0 or 0xFFFF. Originally, implementers used a preset of zero. This preset, however, exposed a weakness in the algorithm. A message that started with an arbitrary number of zeros would have a CRC of zero until a 1 bit was detected. Today, the predominant preset is OxFFFF, which avoids the leading zero problem.

The function **main** (Listing Two) illustrates the behavior of the CCITT-CRC on some test data.

```
/******************************************
//
// main() - test driver for crc16 function
//
*******************************************

```

2014
Dr. Dobb's Journal
November - **Mobile Development**
August - **Web Development**
May - **Testing**
February - **Languages**

Dr. Dobb's Tech Digest
DevOps
Open Source
Windows and .NET programming
The Design of Messaging Middleware and 10 Tips from Tech Writers
Parallel Array Operations in Java 8 and Android on x86: Java Native Interface and the Android Native Development Kit

2013
January - **Mobile Development**
February - **Parallel Programming**
March - **Windows Programming**
April - **Programming Languages**
May - **Web Development**
June - **Database Development**
July - **Testing**
August - **Debugging and Defect Management**
September - **Version Control**
October - **DevOps**
November- **Really Big Data**
December - **Design**

2012
January - **C & C++**
February - **Parallel Programming**
March - **Microsoft Technologies**
April - **Mobile Development**
May - **Database Programming**
June - **Web Development**
July - **Security**
August - **ALM & Development Tools**
September - **Cloud & Web Development**
October - **JVM Languages**
November - **Testing**
December - **DevOps**

2011

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        unsigned short crc;
        static unsigned char string[40];
        string[0] = 'T';
        string[1] = (unsigned char)0xd9;
        string[2] = (unsigned char)0xe4;
        string[3] = NULL;

        printf ("The crc of \"T\" is 0xD9E4.
                       crc16(string, (short)1)

        printf ("The crc of \"T 0xD9 0xE4\"
                       crc16(string, (short)3)

        strcpy(string, "THE,QUICK,BROWN,FOX,
        printf("The crc of \"%s\" is 0x6E20.
                       string, crc1 (string, s

        string[0] = (unsigned char)0x03;
        string[1] = (unsigned char)0x3F;
        puts("CCITT Recommendation X.25 (198
        printf("\tThe crc of 0x03 0x3F is 0x
                       crc16(string, (short)2)

        puts("strike RETURN to continue...")
        getchar();
}
```

Listing Two.

The first case calculates the CRC for the message **T**. The second case calculates the CRC for the message **T** and its CRC. Note that this CRC is a constant, which has been defined in **crc_ok**. (This is the approach usually taken when the CCITT-CRC is implemented in hardware.) The third case illustrates the CRC applied to a longer message. The final case is taken from Appendix I of *CCITT Recommendation X.25*.

A note about portability: I have used the code presented here on four different compilers for three different machine types (8086, 6809, and 68000). You should be able to use it without any problems.

The CCITT-CRC has other applications besides the field of data communications. I used it in an embedded system application to verify the integrity of a block of data held in non-volatile RAM. You could use it to verify any block of data on disk or in memory. And of course you can use it in message protocols of your own devising.

Other Error Detection Schemes

Anytime a message is transferred over a physical medium, the possibility exists that it may be corrupted by noise. Accordingly, since the earliest days of data communication, various mechanisms have been devised to detect when the data received was not the same as the data sent.

One of the simplest error detection schemes is parity checking. Each data byte is sent with an extra bit, which is called the parity bit. The value of the parity bit depends on the number of 1 bits in the byte, and also on the type of parity checking used. When Odd Parity is employed, the parity bit is a 1 when the number of 1 bits in the byte is odd. Otherwise, the parity bit is a 0. When Even Parity is used, a parity bit of 1 indicates an even number of 1 bits. Parity checking is easily implemented in hardware and is a feature found on most data comm chips. Its speed and ease of use make it an attractive and popular error detection mechanism.

Yet, parity checking is rather inefficient. In asynchronous communications, each eight-bit data byte is "framed" by a start bit and a stop bit, for a total of 10 bits. Adding a parity bit to the data byte increases the character size 10 percent. Furthermore, parity checking can only detect an odd number of errors per byte. If two bit errors occur in a single byte, they cancel each other. The parity is unchanged, and the error goes undetected. In an extreme case, all eight data bits in the byte could be reversed, but the parity check would not detect the error.

A more efficient method of detecting errors is the checksum. A checksum is calculated by adding together the values of all of the data bytes in the message. Checksums can be 8, 16, or 32 bits wide (overflow from the addition is ignored). In a typical application, the checksum is appended to the end of the message. The receiver verifies the message by re-calculating the checksum on the data and comparing its result to the checksum that was sent.)

Simple checksums are easy to implement in software and do not bog the processor down. When checksums are employed, data can be transmitted without the overhead of parity bits. This is a consideration that becomes more important as the size of the message increases. However, checksums fall prey to an entire class of errors that can be termed "transposition errors." Imagine that a message is sent containing the sequence 0x31 0x33. With just two bit errors, the sequence could be incorrectly received as 0x33 0x31, and yet still produce a "correct" checksum.