A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS

==
"Everything you wanted to know about CRC algorithms, but were afraid
to ask for fear that errors in your understanding might be detected."

Version : 3.
Date : 19 August 1993.
Author : Ross N. Williams.
Net : ross@guest.adelaide.edu.au.
FTP : ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt
Company : Rocksoft^tm Pty Ltd.
Snail : 16 Lerwick Avenue, Hazelwood Park 5066, Australia.
Fax : +61 8 373-4911 (c/- Internode Systems Pty Ltd).
Phone : +61 8 379-9217 (10am to 10pm Adelaide Australia time).

Contents

PROF

0. Abstract

This document explains CRCs (Cyclic Redundancy Codes) and their table-driven implementations in full, precise detail. Much of the literature on CRCs, and in particular on their table-driven implementations, is a little obscure (or at least seems so to me). This document is an attempt to provide a clear and simple no-nonsense explanation of CRCs and to absolutely nail down every detail of the operation of their high-speed implementations. In addition to this, this document presents a parameterized model CRC algorithm called the "Rocksoft^tm Model CRC Algorithm". The model algorithm can be parameterized to behave like most of the CRC implementations around, and so acts as a good reference for describing particular algorithms. A low-speed implementation of the model CRC algorithm is provided in the C programming language. Lastly there is a section giving two forms of high-speed table driven implementations, and providing a program that generates CRC lookup tables.

1. Introduction: Error Detection

The aim of an error detection technique is to enable the receiver of a message transmitted through a noisy (error-introducing) channel to determine whether the message has been corrupted. To do this, the transmitter constructs a value (called a checksum) that is a function of the message, and appends it to the message. The receiver can then use the same function to calculate the checksum of the received message and compare it with the appended checksum to see if the message was correctly received. For example, if we chose a checksum function which was simply the sum of the bytes in the message mod 256 (i.e. the sum in a one-byte arithmetic), then we could send this checksum with the message. If the sum of the bytes of the received message didn't match the checksum, an error would be detected.

PROF

An error detection technique is good if it meets the following criteria:

1. It should detect the kinds of errors that are likely to occur in the message (some techniques detect some kinds of error better than

others).
2. It should not be too expensive to compute (in terms of time or space), otherwise it will not be practical to use it.
3. The encoding of the checksum should be efficient, as this will affect the length of the message (which in turn affects the probability of an error occurring in the message).

These criteria are often in conflict, so the design of an error detection technique usually involves a tradeoff between the speed of computation, complexity of encoding, and strength of error detection.

The aim of this document is to detail one class of error detection techniques: the CRC (Cyclic Redundancy Check). CRC techniques are widely used (for example in the Ethernet computer networking protocol, and in the CCITT V.42 standard for error correction in modems).

2. The Need For Complexity

In the example given above, a simple sum-of-bytes checksum was described. This kind of checksum is not very reliable. The main problem is that the sum-of-bytes technique does not detect the deletion or insertion of zero bytes, nor does it detect the transposition of words within the message. These problems can be addressed by making the checksum function a little more complicated. For example, if we were to use a sum-of-words instead of a sum-of-bytes, then transpositions would probably generate a different checksum. However, the sum-of-words would still not detect the deletion or insertion of zero words. To make the checksum more sensitive still, we could construct a checksum that was a function not just of the value of each word, but also of its position in the message. For example, we could multiply each word by its word number (1,2,3...) before adding it into the checksum. This would mean that swapping two words would generate a different checksum, and so this kind of error would be detected.

The CRC algorithm is a very powerful and flexible technique that addresses these problems in a robust way. Although CRCs are a little more difficult to understand and implement than a sum-of-words checksum, they have become very popular because they are very good at detecting the kinds of errors that commonly occur, and the computation cost is not great.

3. The Basic Idea Behind CRC Algorithms

The basic idea of a CRC is this: We have a message, which is a sequence of bits. We append some extra bits (the checksum) to the end of the message to form a complete message. When the complete message is received, the receiver can perform a calculation on the complete message and if no errors occurred during transmission, the result of the calculation will be zero. If the result is non-zero, then an error must have occurred.

This sounds a little like the sum-of-bytes example given above, and in a sense it is. The difference is that instead of adding up the bytes with normal arithmetic, the CRC algorithm involves a more complicated calculation based on interpreting the message as a polynomial.

4. Polynomial Arithmetic

To understand CRCs you need to understand a little about polynomial arithmetic, so I will attempt to give a nutshell explanation here.

Imagine that you have a sequence of bits (e.g. 10100110). This can be interpreted as a polynomial in the following way. Each bit represents a coefficient of a term in the polynomial, with the least significant bit representing the coefficient of x^0, the next bit representing the coefficient of x^1, and so on. So the bit sequence 10100110 corresponds to the polynomial:

```
1*x^7 + 0*x^6 + 1*x^5 + 0*x^4 + 0*x^3 + 1*x^2 + 1*x^1 + 0*x^0
```

or equivalently:

```
x^7 + x^5 + x^2 + x
```

PROF

In polynomial arithmetic, addition and subtraction are performed using XOR (exclusive or), because we are only interested in the coefficients modulo 2 (i.e. 0 or 1). For example:

```
x^3 + x + 1
```

XOR x^3 + x^2 + 1
----
x^2 + x

This is because 1 XOR 1 = 0, so the x^3 terms cancel out, and 0 XOR 1 = 1, so the x^2 and x terms remain.

Multiplication and division are a little more complicated, but for the purposes of CRCs, we only need to know that they can be performed, and that division has a remainder (just like normal division).

5. The Mathematics of CRC

Here's how the CRC algorithm works mathematically:

1. Take the message polynomial M(x) (the message bits).
2. Choose a generator polynomial G(x) of degree n (this defines the particular CRC algorithm being used).
3. Multiply M(x) by x^n (i.e. append n zero bits to the message).
4. Divide the result by G(x) and take the remainder R(x).
5. The CRC checksum is R(x) (n bits).
6. The complete message is M(x)*x^n + R(x) (the original message with the CRC appended).

The receiver performs the same division on the complete message, and if no errors have occurred, the remainder will be zero. This is because the complete message was constructed to be divisible by G(x).

6. Choosing A Poly

The generator polynomial G(x) is a critical part of the CRC algorithm, as it determines the kinds of errors that will be detected. The following rules of thumb are used when choosing a polynomial:

1. The polynomial should have a high degree (to detect more errors).
2. It should be irreducible (i.e. it can't be factored into smaller polynomials).
3. It should have a large number of non-zero coefficients (to detect burst errors better).

PROF

A commonly used polynomial is the CRC-32 polynomial used in Ethernet and PKZIP:

```
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
```

This polynomial is known to be very good at detecting errors.

7. A Straightforward CRC Implementation

Here's a straightforward way to implement a CRC algorithm:

1. Initialize a register to all ones (this is standard practice for CRCs).

2. For each bit in the message:
   a. Shift the register left by one bit.
   b. If the bit shifted out was a 1, XOR the register with the generator polynomial.
   c. If the input bit is a 1, XOR the register with the generator polynomial.
3. The final value of the register is the CRC.

This is simple, but slow, as it processes one bit at a time.

8. A Table-Driven Implementation

To speed things up, we can process the message a byte at a time using a lookup table. Here's how it works:

1. Create a table of 256 entries (one for each possible byte).
2. Each entry in the table represents the effect of processing that byte through the straightforward algorithm.
3. To compute the CRC:
   a. Initialize the register to all ones.
   b. For each byte in the message:
   i. Take the current register value and the input byte and use them to index into the table.
   ii. Update the register with the table value.
4. The final register value is the CRC.

This is much faster because it processes a byte at a time instead of a bit.

9. A Clean Table-Driven Implementation

PROF

The table-driven implementation can be made cleaner by combining the register and input byte in a slightly different way. Instead of using the input byte directly to index the table, we XOR it with the top byte of the register, and then shift the register left by 8 bits before XORing with the table value. This eliminates the need to shift the register bit-by-bit.

10. The Final Twist

There's one final optimization that is commonly used. Instead of initializing the register to all ones, we initialize it to zero, and then XOR the final CRC with a constant (usually all ones). This has the same effect as initializing to all ones, but it's faster because we don't need to initialize the register.

11. Reflections

Some CRC implementations use a "reflected" version of the algorithm, where the bits in each byte are reversed (e.g. 10110000 becomes 00001101). This is done to match the way some hardware processes bits (e.g. in serial communications). If you're implementing a CRC to match a particular standard, you need to know whether it uses the reflected or non-reflected version.

12. How to Optimise

Here are some tips for optimizing a CRC implementation:

1. Use a table-driven approach (as described above).
2. Use a larger table (e.g. process 16 bits at a time instead of 8).
3. Use hardware-specific instructions (e.g. CRC instructions on some processors).
4. Unroll loops to reduce overhead.
5. Use parallel processing for very large messages.

13. Security

CRCs are designed for error detection, not security. They are not cryptographically secure, and should not be used to verify the authenticity of a message, as it is relatively easy to construct a message with a given CRC.

14. Summary

PROF

CRCs are a powerful and widely used error detection technique. They are based on polynomial arithmetic, and can be implemented efficiently using table-driven techniques. The choice of polynomial is critical to the error detection properties, and the implementation can be optimized in various ways.

15. A Parameterized Model Algorithm

To make it easier to describe and implement different CRC algorithms, we can define a parameterized model. The Rocksoft^tm Model CRC Algorithm has the following parameters:

- WIDTH: The width of the polynomial (e.g. 32 for CRC-32).
- POLY: The generator polynomial.
- INIT: The initial value of the register.

- REFIN: Whether to reflect the input bytes.
- REFOUT: Whether to reflect the output CRC.
- XOROUT: The value to XOR with the final CRC.

By setting these parameters appropriately, you can emulate most CRC algorithms.

16. The Rocksoft^tm Model CRC Algorithm

Here's the algorithm in pseudocode:

```
FUNCTION CRC (msg, width, poly, init, refin, refout, xorout)
BEGIN
    reg := init
    FOR each byte in msg
        IF refin THEN reflect byte
        reg := (reg >> 8) XOR table[(reg XOR byte) AND 0xFF]
    END FOR
    IF refout THEN reflect reg
    RETURN reg XOR xorout
END
```

17. Implementation of the Rocksoft^tm Model CRC Algorithm

Here's a C implementation of the model algorithm:

```c
typedef unsigned long crc_t;

crc_t crc (unsigned char *msg, int len, int width, crc_t poly,
           crc_t init, int refin, int refout, crc_t xorout)
{
    crc_t reg = init;
    int i;

    for (i = 0; i < len; i++)
    {
        if (refin)
            msg[i] = reflect (msg[i], 8);
        reg = (reg >> 8) ^ crctable[(reg ^ msg[i]) & 0xFF];
    }
    if (refout)
        reg = reflect (reg, width);
    return reg ^ xorout;
}
```

PROF

This assumes the existence of a `reflect` function and a `crctable` array, which would need to be defined separately.

18. Roll Your Own Table-Driven Implementation

To create your own table-driven implementation:

1. Choose your parameters (width, poly, init, refin, refout, xorout).
2. Generate a lookup table based on the polynomial.
3. Use the model algorithm above, substituting your parameters.

19. Generating A Lookup Table

Here's how to generate the lookup table:

```
void make_crctable (crc_t poly, int width, crc_t crctable[256])
{
    int i, j;
    crc_t r;

    for (i = 0; i < 256; i++)
    {
        r = i;
        for (j = 0; j < 8; j++)
            if (r & 1)
                r = (r >> 1) ^ poly;
            else
                r >>= 1;
        crctable[i] = r;
    }
}
```

PROF

This assumes the polynomial is already in the correct form (i.e. reflected if necessary).

20. Summary

This document has provided a detailed explanation of CRC algorithms, including their mathematical basis, implementation techniques, and optimizations. The Rocksoft^tm Model CRC Algorithm provides a flexible way to describe and implement most CRC algorithms.

21. Corrections

If you find any errors in this document, please contact the author.

A. Glossary

- **Checksum**: A value calculated from a message to detect errors.
- **Polynomial**: A mathematical expression used in CRC calculations.
- **Generator Polynomial**: The polynomial used to define a particular CRC algorithm.
- **Reflection**: Reversing the bits in a byte or word.

B. References

[Knuth81] Knuth, D.E., "The Art of Computer Programming", Volume 2: Seminumerical Algorithms, Section 4.6.
[Nelson 91] Nelson, M., "The Data Compression Book", M&T Books, (501 Galveston Drive, Redwood City, CA 94063), 1991, ISBN: 1-55851-214-4.
Comment: If you want to see a real implementation of a real 32-bit checksum algorithm, look on pages 440, and 446-448.
[Sarwate88] Sarwate, D.V., "Computation of Cyclic Redundancy Checks via Table Look-Up", Communications of the ACM, 31(8), pp.1008-1013.
Comment: This paper describes a high-speed table-driven implementation for CRC algorithms that is superior to the tea-leaf algorithm. Although this paper describes the technique used by most modern CRC implementations, I found the appendix of this paper (where all the good stuff is) difficult to understand.
[Tanenbaum81] Tanenbaum, A.S., "Computer Networks", Prentice Hall, 1981, ISBN: 0-13-164699-0.
Comment: Section 3.5.3 on pages 128 to 132 provides a very clear description of CRC codes. However, it does not describe table-driven implementation techniques.

PROF

C. References I Have Detected But Haven't Yet Sighted

Boudreau, Steen, "Cyclic Redundancy Checking by Program," AFIPS Proceedings, Vol. 39, 1971.

Davies, Barber, "Computer Networks and Their Protocols," J. Wiley & Sons, 1979.

Higginson, Kirstein, "On the Computation of Cyclic Redundancy Checks by Program," The Computer Journal (British), Vol. 16, No. 1, Feb 1973.

McNamara, J. E., "Technical Aspects of Data Communication," 2nd Edition, Digital Press, Bedford, Massachusetts, 1982.

Marton and Frambs, "A Cyclic Redundancy Checking (CRC) Algorithm," Honeywell Computer Journal, Vol. 5, No. 3, 1971.

Nelson M., "File verification using CRC", Dr Dobbs Journal, May 1992, pp.64-67.