Acquired from:

    ftp.adelaide.edu.au:/pub/rocksoft/crc_v3.txt
    or ftp://ftp.rocksoft.com/papers/crc_v3.txt
    or http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

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."

Table of Contents

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. modulo 256), then it might go something as follows. All numbers
are in decimal.

    Message   : 6 23 4
    Message with checksum   : 6 23 4 33
    Message after transmission : 6 27 4 33

In the above, the second byte of the message was corrupted from 23 to
27 by the communications channel. However, the receiver can detect
this by comparing the transmitted checksum (33) with the computer
checksum of 37 (6 + 27 + 4). If the checksum itself is corrupted, a
correctly transmitted message might be incorrectly identified as a
corrupted one. However, this is a safe-side failure. A dangerous-side
failure occurs where the message and/or checksum is corrupted in a
manner that results in a transmission that is internally consistent.
Unfortunately, this possibility is completely unavoidable and the best
that can be done is to minimize its probability by increasing the
amount of information in the checksum (e.g. widening the checksum from
one byte to two bytes).

Other error detection techniques exist that involve performing complex
transformations on the message to inject it with redundant
information. However, this document addresses only CRC algorithms,
which fall into the class of error detection algorithms that leave the
data intact and append a checksum on the end. i.e.:

     <original intact message> <checksum>

2. The Need For Complexity

In the checksum example in the previous section, we saw how a
corrupted message was detected using a checksum algorithm that simply
sums the bytes in the message mod 256:

```
   Message                   :  6 23  4
   Message with checksum     :  6 23  4 33
   Message after transmission :  6 27  4 33
```

A problem with this algorithm is that it is too simple. If a number of
random corruptions occur, there is a 1 in 256 chance that they will
not be detected. For example:

```
   Message                   :  6 23  4
   Message with checksum     :  6 23  4 33
   Message after transmission :  8 20  5 33
```

To strengthen the checksum, we could change from an 8-bit register to
a 16-bit register (i.e. sum the bytes mod 65536 instead of mod 256) so
as to apparently reduce the probability of failure from 1/256 to
1/65536. While basically a good idea, it fails in this case because
the formula used is not sufficiently "random"; with a simple summing
formula, each incoming byte affects roughly only one byte of the
summing register no matter how wide it is. For example, in the second
example above, the summing register could be a Megabyte wide, and the
error would still go undetected. This problem can only be solved by
replacing the simple summing formula with a more sophisticated formula
that causes each incoming byte to have an effect on the entire
checksum register.

Thus, we see that at least two aspects are required to form a strong
checksum function:

   WIDTH: A register width wide enough to provide a low a-priori
          probability of failure (e.g. 32-bits gives a 1/2^32 chance
          of failure).

   CHAOS: A formula that gives each input byte the potential to change
          any number of bits in the register.

Note: The term "checksum" was presumably used to describe early
summing formulas, but has now taken on a more general meaning
encompassing more sophisticated algorithms such as the CRC ones. The
CRC algorithms to be described satisfy the second condition very well,
and can be configured to operate with a variety of checksum widths.

3. The Basic Idea Behind CRC Algorithms

Where might we go in our search for a more complex function than
summing? All sorts of schemes spring to mind. We could construct
tables using the digits of pi, or hash each incoming byte with all the
bytes in the register. We could even keep a large telephone book
on-line, and use each incoming byte combined with the register bytes
to index a new phone number which would be the next register value.
The possibilities are limitless.

However, we do not need to go so far; the next arithmetic step
suffices. While addition is clearly not strong enough to form an
effective checksum, it turns out that division is, so long as the

divisor is about as wide as the checksum register.

The basic idea of CRC algorithms is simply to treat the message as an
enormous binary number, to divide it by another fixed binary number,
and to make the remainder from this division the checksum. Upon
receipt of the message, the receiver can perform the same division and
compare the remainder with the "checksum" (transmitted remainder).

Example: Suppose the the message consisted of the two bytes (6,23) as
in the previous example. These can be considered to be the hexadecimal
number 0617 which can be considered to be the binary number
0000-0110-0001-0111. Suppose that we use a checksum register one-byte
wide and use a constant divisor of 1001, then the checksum is the
remainder after 0000-0110-0001-0111 is divided by 1001. While in this
case, this calculation could obviously be performed using common
garden variety 32-bit registers, in the general case this is messy. So
instead, we'll do the division using good-'ol long division which you
learnt in school (remember?). Except this time, it's in binary:

```
          ...0000010101101 = 00AD =  173 = QUOTIENT
             ____-___-___-___-
9= 1001 )  0000011000010111 = 0617 = 1559 = DIVIDEND
DIVISOR    0000.,,....,.,,,
           ----.,,....,.,,,
            0000,,....,.,,,
            0000,,....,.,,,
            ----,,....,.,,,
             0001,....,.,,,
             0000,....,.,,,
             ----,....,.,,,
              0011....,.,,,
              0000....,.,,,
              ----....,.,,,
               0110...,.,,,
               0000...,.,,,
               ----...,.,,,
                1100..,.,,,
                1001..,.,,,
                ====..,.,,,
                 0110.,.,,,
                 0000.,.,,,
                 ----.,.,,,
                  1100,.,,,
                  1001,.,,,
                  ====,.,,,
                   0111.,,,
                   0000.,,,
                   ----.,,,
                    1110,,,
                    1001,,,
                    ====,,,
                     1011,,
                     1001,,
                     ====,,
                      0101,
                      0000,
                      ----
                       1011
                       1001
                       ====
```

                          0010 = 02 = 2 = REMAINDER

In decimal this is "1559 divided by 9 is 173 with a remainder of 2".

Although the effect of each bit of the input message on the quotient
is not all that significant, the 4-bit remainder gets kicked about
quite a lot during the calculation, and if more bytes were added to
the message (dividend) it's value could change radically again very
quickly. This is why division works where addition doesn't.

In case you're wondering, using this 4-bit checksum the transmitted
message would look like this (in hexadecimal): 06172 (where the 0617
is the message and the 2 is the checksum). The receiver would divide
0617 by 9 and see whether the remainder was 2.

4. Polynomical Arithmetic

While the division scheme described in the previous section is very
very similar to the checksumming schemes called CRC schemes, the CRC
schemes are in fact a bit weirder, and we need to delve into some
strange number systems to understand them.

The word you will hear all the time when dealing with CRC algorithms
is the word "polynomial". A given CRC algorithm will be said to be
using a particular polynomial, and CRC algorithms in general are said
to be operating using polynomial arithmetic. What does this mean?

Instead of the divisor, dividend (message), quotient, and remainder
(as described in the previous section) being viewed as positive
integers, they are viewed as polynomials with binary coefficients.
This is done by treating each number as a bit-string whose bits are
the coefficients of a polynomial. For example, the ordinary number 23
(decimal) is 17 (hex) and 10111 binary and so it corresponds to the
polynomial:

    $1*x^4 + 0*x^3 + 1*x^2 + 1*x^1 + 1*x^0$

or, more simply:

    $x^4 + x^2 + x^1 + x^0$

Using this technique, the message, and the divisor can be represented
as polynomials and we can do all our arithmetic just as before, except
that now it's all cluttered up with Xs. For example, suppose we wanted
to multiply 1101 by 1011. We can do this simply by multiplying the
polynomials:

$(x^3 + x^2 + x^0)(x^3 + x^1 + x^0)$
$= (x^6 + x^4 + x^3$
 $+ x^5 + x^3 + x^2$
 $+ x^3 + x^1 + x^0) = x^6 + x^5 + x^4 + 3*x^3 + x^2 + x^1 + x^0$

At this point, to get the right answer, we have to pretend that x is 2
and propagate binary carries from the $3*x^3$ yielding

    $x^7 + x^3 + x^2 + x^1 + x^0$

It's just like ordinary arithmetic except that the base is abstracted

and brought into all the calculations explicitly instead of being
there implicitly. So what's the point?

The point is that IF we pretend that we DON'T know what x is, we CAN'T
perform the carries. We don't know that $3*x^3$ is the same as $x^4 + x^3$
because we don't know that x is 2. In this true polynomial arithmetic
the relationship between all the coefficients is unknown and so the
coefficients of each power effectively become strongly typed;
coefficients of x^2 are effectively of a different type to
coefficients of x^3.

With the coefficients of each power nicely isolated, mathematicians
came up with all sorts of different kinds of polynomial arithmetics
simply by changing the rules about how coefficients work. Of these
schemes, one in particular is relevant here, and that is a polynomial
arithmetic where the coefficients are calculated MOD 2 and there is no
carry; all coefficients must be either 0 or 1 and no carries are
calculated. This is called "polynomial arithmetic mod 2". Thus,
returning to the earlier example:

$(x^3 + x^2 + x^0)(x^3 + x^1 + x^0)$
$= (x^6 + x^4 + x^3$
$ + x^5 + x^3 + x^2$
$ + x^3 + x^1 + x^0)$
$= x^6 + x^5 + x^4 + 3*x^3 + x^2 + x^1 + x^0$

Under the other arithmetic, the $3*x^3$ term was propagated using the
carry mechanism using the knowledge that x=2. Under "polynomial
arithmetic mod 2", we don't know what x is, there are no carries, and
all coefficients have to be calculated mod 2. Thus, the result
becomes:

$= x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0$

As Knuth [Knuth81] says (p.400):

    "The reader should note the similarity between polynomial
    arithmetic and multiple-precision arithmetic (Section 4.3.1), where
    the radix b is substituted for x. The chief difference is that the
    coefficient u_k of x^k in polynomial arithmetic bears little or no
    relation to its neighboring coefficients x^{k-1} [and x^{k+1}], so
    the idea of "carrying" from one place to another is absent. In fact
    polynomial arithmetic modulo b is essentially identical to multiple
    precision arithmetic with radix b, except that all carries are
    suppressed."

Thus polynomical arithmetic mod 2 is just binary arithmetic mod 2 with
no carries. While polynomials provide useful mathematical machinery in
more analytical approaches to CRC and error-correction algorithms, for
the purposes of exposition they provide no extra insight and some
encumbrance and have been discarded in the remainder of this document
in favour of direct manipulation of the arithmetical system with which
they are isomorphic: binary arithmetic with no carry.

5. Binary Arithmetic with No Carries

Having dispensed with polynomials, we can focus on the real arithmetic
issue, which is that all the arithmetic performed during CRC
calculations is performed in binary with no carries. Often this is

called polynomial arithmetic, but as I have declared the rest of this
document a polynomial free zone, we'll have to call it CRC arithmetic
instead. As this arithmetic is a key part of CRC calculations, we'd
better get used to it. Here we go:

Adding two numbers in CRC arithmetic is the same as adding numbers in
ordinary binary arithmetic except there is no carry. This means that
each pair of corresponding bits determine the corresponding output bit
without reference to any other bit positions. For example:

```
      10011011
     +11001010
      --------
      01010001
      --------
```

There are only four cases for each bit position:

```
   0+0=0
   0+1=1
   1+0=1
   1+1=0  (no carry)
```

Subtraction is identical:

```
      10011011
     -11001010
      --------
      01010001
      --------
```

with

```
   0-0=0
   0-1=1  (wraparound)
   1-0=1
   1-1=0
```

In fact, both addition and subtraction in CRC arithmetic is equivalent
to the XOR operation, and the XOR operation is its own inverse. This
effectively reduces the operations of the first level of power
(addition, subtraction) to a single operation that is its own inverse.
This is a very convenient property of the arithmetic.

By collapsing of addition and subtraction, the arithmetic discards any
notion of magnitude beyond the power of its highest one bit. While it
seems clear that 1010 is greater than 10, it is no longer the case
that 1010 can be considered to be greater than 1001. To see this, note
that you can get from 1010 to 1001 by both adding and subtracting the
same quantity:

```
   1010 = 1010 + 0011
   1010 = 1010 - 0011
```

This makes nonsense of any notion of order.

Having defined addition, we can move to multiplication and division.
Multiplication is absolutely straightforward, being the sum of the
first number, shifted in accordance with the second number.

```
        1101
      x 1011
        ----
        1101
       1101.
      0000..
     1101...
     -------
     1111111   Note: The sum uses CRC addition
     -------
```

Division is a little messier as we need to know when "a number goes
into another number". To do this, we invoke the weak definition of
magnitude defined earlier: that X is greater than or equal to Y iff
the position of the highest 1 bit of X is the same or greater than the
position of the highest 1 bit of Y. Here's a fully worked division
(nicked from [Tanenbaum81]).

```
              1100001010
            _____
    10011 ) 11010110110000
            10011,,.,,....
            -----,,.,,....
             10011,.,,....
             10011,.,,....
             -----,.,,....
              00001.,,....
              00000.,,....
              -----.,,....
               00010,,....
               00000,,....
               -----,,....
                00101,....
                00000,....
                -----,....
                 01011....
                 00000....
                 -----....
                  10110...
                  10011...
                  -----...
                   01010..
                   00000..
                   -----..
                    10100.
                    10011.
                    -----.
                     01110
                     00000
                     -----
                      1110 = Remainder
```

That's really it. Before proceeding further, however, it's worth our
while playing with this arithmetic a bit to get used to it.

We've already played with addition and subtraction, noticing that they
are the same thing. Here, though, we should note that in this
arithmetic A+0=A and A-0=A. This obvious property is very useful
later.

In dealing with CRC multiplication and division, it's worth getting a
feel for the concepts of MULTIPLE and DIVISIBLE.

If a number A is a multiple of B then what this means in CRC
arithmetic is that it is possible to construct A from zero by XORing
in various shifts of B. For example, if A was 0111010110 and B was 11,
we could construct A from B as follows:

```
          0111010110
      = .......11.
      + ....11....
      + ...11.....
        .11.......
```

However, if A is 0111010111, it is not possible to construct it out of
various shifts of B (can you see why? - see later) so it is said to be
not divisible by B in CRC arithmetic.

Thus we see that CRC arithmetic is primarily about XORing particular
values at various shifting offsets.

6. A Fully Worked Example

Having defined CRC arithmetic, we can now frame a CRC calculation as
simply a division, because that's all it is! This section fills in the
details and gives an example.

To perform a CRC calculation, we need to choose a divisor. In maths
marketing speak the divisor is called the "generator polynomial" or
simply the "polynomial", and is a key parameter of any CRC algorithm.
It would probably be more friendly to call the divisor something else,
but the poly talk is so deeply ingrained in the field that it would
now be confusing to avoid it. As a compromise, we will refer to the
CRC polynomial as the "poly". Just think of this number as a sort of
parrot. "Hello poly!"

You can choose any poly and come up with a CRC algorithm. However,
some polys are better than others, and so it is wise to stick with the
tried an tested ones. A later section addresses this issue.

The width (position of the highest 1 bit) of the poly is very
important as it dominates the whole calculation. Typically, widths of
16 or 32 are chosen so as to simplify implementation on modern
computers. The width of a poly is the actual bit position of the
highest bit. For example, the width of 10011 is 4, not 5. For the
purposes of example, we will chose a poly of 10011 (of width W of 4).

Having chosen a poly, we can proceed with the calculation. This is
simply a division (in CRC arithmetic) of the message by the poly. The
only trick is that W zero bits are appended to the message before the
CRC is calculated. Thus we have:

```
   Original message                 : 1101011011
   Poly                             :      10011
   Message after appending W zeros : 11010110110000
```

Now we simply divide the augmented message by the poly using CRC
arithmetic. This is the same division as before:

```
            1100001010 = Quotient (nobody cares about the quotient)
          _____
10011 )  11010110110000 = Augmented message (1101011011 + 0000)
=Poly  10011,,.,,....
        -----,,.,,....
         10011,.,,....
         10011,.,,....
        -----,.,,....
         00001.,,....
         00000.,,....
        -----.,,....
          00010,,....
          00000,,....
         -----,,....
           00101,....
           00000,....
          -----,....
            01011....
            00000....
           -----....
             10110...
             10011...
            -----...
              01010..
              00000..
             -----..
               10100.
               10011.
              -----.
                01110
                00000
               -----
                 1110 = Remainder = THE CHECKSUM!!!!
```

The division yields a quotient, which we throw away, and a remainder,
which is the calculated checksum. This ends the calculation.

Usually, the checksum is then appended to the message and the result
transmitted. In this case the transmission would be: 11010110111110.

At the other end, the receiver can do one of two things:

    a. Separate the message and checksum. Calculate the checksum for
       the message (after appending W zeros) and compare the two
       checksums.

    b. Checksum the whole lot (without appending zeros) and see if it
       comes out as zero!

These two options are equivalent. However, in the next section, we
will be assuming option b because it is marginally mathematically
cleaner.

A summary of the operation of the class of CRC algorithms:

    1. Choose a width W, and a poly G (of width W).
    2. Append W zero bits to the message. Call this M'.
    3. Divide M' by G using CRC arithmetic. The remainder is the checksum.

That's all there is to it.

7. Choosing A Poly

Choosing a poly is somewhat of a black art and the reader is referred
to [Tanenbaum81] (p.130-132) which has a very clear discussion of this
issue. This section merely aims to put the fear of death into anyone
who so much as toys with the idea of making up their own poly. If you
don't care about why one poly might be better than another and just
want to find out about high-speed implementations, choose one of the
arithmetically sound polys listed at the end of this section and skip
to the next section.

First note that the transmitted message T is a multiple of the poly.
To see this, note that 1) the last W bits of T is the remainder after
dividing the augmented (by zeros remember) message by the poly, and 2)
addition is the same as subtraction so adding the remainder pushes the
value up to the next multiple. Now note that if the transmitted
message is corrupted in transmission that we will receive T+E where E
is an error vector (and + is CRC addition (i.e. XOR)). Upon receipt of
this message, the receiver divides T+E by G. As T mod G is 0, (T+E)
mod G = E mod G. Thus, the capacity of the poly we choose to catch
particular kinds of errors will be determined by the set of multiples
of G, for any corruption E that is a multiple of G will be undetected.
Our task then is to find classes of G whose multiples look as little
like the kind of line noise (that will be creating the corruptions) as
possible. So let's examine the kinds of line noise we can expect.

SINGLE BIT ERRORS: A single bit error means E=1000...0000. We can
ensure that this class of error is always detected by making sure that
G has at least two bits set to 1. Any multiple of G will be
constructed using shifting and adding and it is impossible to
construct a value with a single bit by shifting an adding a single
value with more than one bit set, as the two end bits will always
persist.

TWO-BIT ERRORS: To detect all errors of the form 100...000100...000
(i.e. E contains two 1 bits) choose a G that does not have multiples
that are 11, 101, 1001, 10001, 100001, etc. It is not clear to me how
one goes about doing this (I don't have the pure maths background),
but Tanenbaum assures us that such G do exist, and cites G with 1 bits
(15,14,1) turned on as an example of one G that won't divide anything
less than 1...1 where ... is 32767 zeros.

ERRORS WITH AN ODD NUMBER OF BITS: We can catch all corruptions where
E has an odd number of bits by choosing a G that has an even number of
bits. To see this, note that 1) CRC multiplication is simply XORing a
constant value into a register at various offsets, 2) XORing is simply
a bit-flip operation, and 3) if you XOR a value with an even number of
bits into a register, the oddness of the number of 1 bits in the
register remains invariant. Example: Starting with E=111, attempt to
flip all three bits to zero by the repeated application of XORing in
11 at one of the two offsets (i.e. "E=E XOR 011" and "E=E XOR 110")
This is nearly isomorphic to the "glass tumblers" party puzzle where
you challenge someone to flip three tumblers by the repeated
application of the operation of flipping any two. Most of the popular
CRC polys contain an even number of 1 bits. (Note: Tanenbaum states
more specifically that all errors with an odd number of bits can be
caught by making G a multiple of 11).

BURST ERRORS: A burst error looks like E=000...000111...11110000...00.

That is, E consists of all zeros except for a run of 1s somewhere inside. This can be recast as E=(10000...00)(1111111...111) where there are z zeros in the LEFT part and n ones in the RIGHT part. To catch errors of this kind, we simply set the lowest bit of G to 1. Doing this ensures that LEFT cannot be a factor of G. Then, so long as G is wider than RIGHT, the error will be detected. See Tanenbaum for a clearer explanation of this; I'm a little fuzzy on this one. Note: Tanenbaum asserts that the probability of a burst of length greater than W getting through is $(0.5)^W$.

That concludes the section on the fine art of selecting polys.

Some popular polys are:
```
16 bits: (16,12,5,0)                              [X25 standard]
         (16,15,2,0)                              ["CRC-16"]
32 bits: (32,26,23,22,16,12,11,10,8,7,5,4,2,1,0)  [Ethernet]
```

8. A Straightforward CRC Implementation

That's the end of the theory; now we turn to implementations. To start with, we examine an absolutely straight-down-the-middle boring straightforward low-speed implementation that doesn't use any speed tricks at all. We'll then transform that program progessively until we end up with the compact table-driven code we all know and love and which some of us would like to understand.

To implement a CRC algorithm all we have to do is implement CRC division. There are two reasons why we cannot simply use the divide instruction of whatever machine we are on. The first is that we have to do the divide in CRC arithmetic. The second is that the dividend might be ten megabytes long, and todays processors do not have registers that big.

So to implement CRC division, we have to feed the message through a division register. At this point, we have to be absolutely precise about the message data. In all the following examples the message will be considered to be a stream of bytes (each of 8 bits) with bit 7 of each byte being considered to be the most significant bit (MSB). The bit stream formed from these bytes will be the bit stream with the MSB (bit 7) of the first byte first, going down to bit 0 of the first byte, and then the MSB of the second byte and so on.

With this in mind, we can sketch an implementation of the CRC division. For the purposes of example, consider a poly with W=4 and the poly=10111. Then, the perform the division, we need to use a 4-bit register:

```
                  3   2   1   0   Bits
               +---+---+---+---+
       Pop! <-- |   |   |   |   | <----- Augmented message
               +---+---+---+---+

         1   0   1   1   1   = The Poly
```

(Reminder: The augmented message is the message followed by W zero bits.)

To perform the division perform the following:

    Load the register with zero bits.

```
   Augment the message by appending W zero bits to the end of it.
   While (more message bits)
      Begin
      Shift the register left by one bit, reading the next bit of the
         augmented message into register bit position 0.
      If (a 1 bit popped out of the register during step 3)
         Register = Register XOR Poly.
      End
   The register now contains the remainder.
```

(Note: In practice, the IF condition can be tested by testing the top
 bit of R before performing the shift.)

We will call this algorithm "SIMPLE".

This might look a bit messy, but all we are really doing is
"subtracting" various powers (i.e. shiftings) of the poly from the
message until there is nothing left but the remainder. Study the
manual examples of long division if you don't understand this.

It should be clear that the above algorithm will work for any width W.

9. A Table-Driven Implementation

The SIMPLE algorithm above is a good starting point because it
corresponds directly to the theory presented so far, and because it is
so SIMPLE. However, because it operates at the bit level, it is rather
awkward to code (even in C), and inefficient to execute (it has to
loop once for each bit). To speed it up, we need to find a way to
enable the algorithm to process the message in units larger than one
bit. Candidate quantities are nibbles (4 bits), bytes (8 bits), words
(16 bits) and longwords (32 bits) and higher if we can achieve it. Of
these, 4 bits is best avoided because it does not correspond to a byte
boundary. At the very least, any speedup should allow us to operate at
byte boundaries, and in fact most of the table driven algorithms
operate a byte at a time.

For the purposes of discussion, let us switch from a 4-bit poly to a
32-bit one. Our register looks much the same, except the boxes
represent bytes instead of bits, and the Poly is 33 bits (one implicit
1 bit at the top and 32 "active" bits) (W=32).

```
                  3    2    1    0   Bytes
               +----+----+----+----+
      Pop! <-- |    |    |    |    | <----- Augmented message
               +----+----+----+----+

               1<------32 bits------>
```

The SIMPLE algorithm is still applicable. Let us examine what it does.
Imagine that the SIMPLE algorithm is in full swing and consider the
top 8 bits of the 32-bit register (byte 3) to have the values:

   t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0

In the next iteration of SIMPLE, t_7 will determine whether the Poly
will be XORed into the entire register. If t_7=1, this will happen,
otherwise it will not. Suppose that the top 8 bits of the poly are g_7
g_6.. g_0, then after the next iteration, the top byte will be:

```
       t6 t5 t4 t3 t2 t1 t0 ??
+ t7 * (g7 g6 g5 g4 g3 g2 g1 g0)     [Reminder: + is XOR]
```

The NEW top bit (that will control what happens in the next iteration)
now has the value t6 + t7*g7. The important thing to notice here is
that from an informational point of view, all the information required
to calculate the NEW top bit was present in the top TWO bits of the
original top byte. Similarly, the NEXT top bit can be calculated in
advance SOLELY from the top THREE bits t7, t6, and t5. In fact, in
general, the value of the top bit in the register in k iterations can
be calculated from the top k bits of the register. Let us take this
for granted for a moment.

Consider for a moment that we use the top 8 bits of the register to
calculate the value of the top bit of the register during the next 8
iterations. Suppose that we drive the next 8 iterations using the
calculated values (which we could perhaps store in a single byte
register and shift out to pick off each bit). Then we note three
things:

   * The top byte of the register now doesn't matter. No matter how
     many times and at what offset the poly is XORed to the top 8
     bits, they will all be shifted out the right hand side during the
     next 8 iterations anyway.

   * The remaining bits will be shifted left one position and the
     rightmost byte of the register will be shifted in the next byte

   AND

   * While all this is going on, the register will be subjected to a
     series of XOR's in accordance with the bits of the pre-calculated
     control byte.

Now consider the effect of XORing in a constant value at various
offsets to a register. For example:

       0100010 Register
       ...0110 XOR this
       ..0110. XOR this
       0110... XOR this

       0011000

The point of this is that you can XOR constant values into a register
to your heart's delight, and in the end, there will exist a value
which when XORed in with the original register will have the same
effect as all the other XORs.

Perhaps you can see the solution now. Putting all the pieces together
we have an algorithm that goes like this:

   While (augmented message is not exhausted)
      Begin
      Examine the top byte of the register
      Calculate the control byte from the top byte of the register
      Sum all the Polys at various offsets that are to be XORed into
```

```
        the register in accordance with the control byte
    Shift the register left by one byte, reading a new message byte
        into the rightmost byte of the register
    XOR the summed polys to the register
    End
```

As it stands this is not much better than the SIMPLE algorithm.
However, it turns out that most of the calculation can be precomputed
and assembled into a table. As a result, the above algorithm can be
reduced to:

```
    While (augmented message is not exhaused)
        Begin
        Top = top_byte(Register);
        Register = (Register << 24) | next_augmessage_byte;
        Register = Register XOR precomputed_table[Top];
        End
```

There! If you understand this, you've grasped the main idea of
table-driven CRC algorithms. The above is a very efficient algorithm
requiring just a shift, and OR, an XOR, and a table lookup per byte.
Graphically, it looks like this:

```
                      3    2    1    0   Bytes
                   +----+----+----+----+
        +-----<|    |    |    |    | <----- Augmented message
        |       +----+----+----+----+
        |                ^
        |                |
        |               XOR
        |                |
        |      0+----+----+----+----+            Algorithm
        v       +----+----+----+----+            ---------
        |       +----+----+----+----+        1. Shift the register left by
        |       +----+----+----+----+           one byte, reading in a new
        |       +----+----+----+----+           message byte.
        |       +----+----+----+----+        2. Use the top byte just rotated
        |       +----+----+----+----+           out of the register to index
        +----->+----+----+----+----+           the table of 256 32-bit values.
                +----+----+----+----+        3. XOR the table value into the
                +----+----+----+----+           register.
                +----+----+----+----+        4. Goto 1 iff more augmented
                +----+----+----+----+           message bytes.
           255+----+----+----+----+
```

In C, the algorithm main loop looks like this:

```
    r=0;
    while (len--)
      {
      byte t = (r >> 24) & 0xFF;
      r = (r << 8) | *p++;
      r^=table[t];
      }
```

where len is the length of the augmented message in bytes, p points to
the augmented message, r is the register, t is a temporary, and table
is the computed table. This code can be made even more unreadable as
follows:

```
    r=0; while (len--) r = ((r << 8) | *p++) ^ t[(r >> 24) & 0xFF];
```

This is a very clean, efficient loop, although not a very obvious one
to the casual observer not versed in CRC theory. We will call this the
TABLE algorithm.

10. A Slightly Mangled Table-Driven Implementation

Despite the terse beauty of the line

```
    r=0; while (len--) r = ((r << 8) | *p++) ^ t[(r >> 24) & 0xFF];
```

those optimizing hackers couldn't leave it alone. The trouble, you
see, is that this loop operates upon the AUGMENTED message and in
order to use this code, you have to append W/8 zero bytes to the end
of the message before pointing p at it. Depending on the run-time
environment, this may or may not be a problem; if the block of data
was handed to us by some other code, it could be a BIG problem. One
alternative is simply to append the following line after the above
loop, once for each zero byte:

```
    for (i=0; i<W/4; i++) r = (r << 8) ^ t[(r >> 24) & 0xFF];
```

This looks like a sane enough solution to me. However, at the further
expense of clarity (which, you must admit, is already a pretty scare
commodity in this code) we can reorganize this small loop further so
as to avoid the need to either augment the message with zero bytes, or
to explicitly process zero bytes at the end as above. To explain the
optimization, we return to the processing diagram given earlier.

```
                    3    2    1    0   Bytes
                 +----+----+----+----+
          +-----<|    |    |    |    | <----- Augmented message
          |      +----+----+----+----+
          |                ^
          |                |
          |               XOR
          |                |
          |     0+----+----+----+----+         Algorithm
          v      +----+----+----+----+         ---------
          |      +----+----+----+----+      1. Shift the register left by
          |      +----+----+----+----+         one byte, reading in a new
          |      +----+----+----+----+         message byte.
          |      +----+----+----+----+      2. Use the top byte just rotated
          |      +----+----+----+----+         out of the register to index
          +----->+----+----+----+----+         the table of 256 32-bit values.
                 +----+----+----+----+      3. XOR the table value into the
                 +----+----+----+----+         register.
                 +----+----+----+----+      4. Goto 1 iff more augmented
                 +----+----+----+----+         message bytes.
              255+----+----+----+----+
```

Now, note the following facts:

TAIL: The W/4 augmented zero bytes that appear at the end of the
      message will be pushed into the register from the right as all
      the other bytes are, but their values (0) will have no effect
      whatsoever on the register because 1) XORing with zero does not

change the target byte, and 2) the four bytes are never
propagated out the left side of the register where their
zeroness might have some sort of influence. Thus, the sole
function of the W/4 augmented zero bytes is to drive the
calculation for another W/4 byte cycles so that the end of the
REAL data passes all the way through the register.

HEAD: If the initial value of the register is zero, the first four
      iterations of the loop will have the sole effect of shifting in
      the first four bytes of the message from the right. This is
      because the first 32 control bits are all zero and so nothing is
      XORed into the register. Even if the initial value is not zero,
      the first 4 byte iterations of the algorithm will have the sole
      effect of shifting the first 4 bytes of the message into the
      register and then XORing them with some constant value (that is
      a function of the initial value of the register).

These facts, combined with the XOR property

    (A xor B) xor C = A xor (B xor C)

mean that message bytes need not actually travel through the W/4 bytes
of the register. Instead, they can be XORed into the top byte just
before it is used to index the lookup table. This leads to the
following modified version of the algorithm.

```
    +-----<Message (non augmented)
    |
    v         3    2    1    0   Bytes
    |      +----+----+----+----+
  XOR----<|    |    |    |    |
    |      +----+----+----+----+
    |                ^
    |                |
    |               XOR
    |                |
    |     0+----+----+----+----+          Algorithm
    v      +----+----+----+----+          ---------
    |      +----+----+----+----+          1. Shift the register left by
    |      +----+----+----+----+             one byte, reading in a new
    |      +----+----+----+----+             message byte.
    |      +----+----+----+----+          2. XOR the top byte just rotated
    |      +----+----+----+----+             out of the register with the
    +----->+----+----+----+----+             next message byte to yield an
           +----+----+----+----+             index into the table ([0,255]).
           +----+----+----+----+          3. XOR the table value into the
           +----+----+----+----+             register.
           +----+----+----+----+          4. Goto 1 iff more augmented
       255+----+----+----+----+             message bytes.
```

Note: The initial register value for this algorithm must be the
initial value of the register for the previous algorithm fed through
the table four times. Note: The table is such that if the previous
algorithm used 0, the new algorithm will too.

This is an IDENTICAL algorithm and will yield IDENTICAL results. The C
code looks something like this:

```
    r=0; while (len--) r = (r<<8) ^ t[(r >> 24) ^ *p++];
```

and THIS is the code that you are likely to find inside current
table-driven CRC implementations. Some FF masks might have to be ANDed
in here and there for portability's sake, but basically, the above
loop is IT. We will call this the DIRECT TABLE ALGORITHM.

During the process of trying to understand all this stuff, I managed
to derive the SIMPLE algorithm and the table-driven version derived
from that. However, when I compared my code with the code found in
real-implementations, I was totally bamboozled as to why the bytes
were being XORed in at the wrong end of the register! It took quite a
while before I figured out that theirs and my algorithms were actually
the same. Part of why I am writing this document is that, while the
link between division and my earlier table-driven code is vaguely
apparent, any such link is fairly well erased when you start pumping
bytes in at the "wrong end" of the register. It looks all wrong!

If you've got this far, you not only understand the theory, the
practice, the optimized practice, but you also understand the real
code you are likely to run into. Could get any more complicated? Yes
it can.

11. "Reflected" Table-Driven Implementations

Despite the fact that the above code is probably optimized about as
much as it could be, this did not stop some enterprising individuals
from making things even more complicated. To understand how this
happened, we have to enter the world of hardware.

DEFINITION: A value/register is reflected if it's bits are swapped
around its centre. For example: 0101 is the 4-bit reflection of 1010.
0011 is the reflection of 1100.
0111-0101-1010-1111-0010-0101-1011-1100 is the reflection of
0011-1101-1010-0100-1111-0101-1010-1110.

Turns out that UARTs (those handy little chips that perform serial IO)
are in the habit of transmitting each byte with the least significant
bit (bit 0) first and the most significant bit (bit 7) last (i.e.
reflected). An effect of this convention is that hardware engineers
constructing hardware CRC calculators that operate at the bit level
took to calculating CRCs of bytes streams with each of the bytes
reflected within itself. The bytes are processed in the same order,
but the bits in each byte are swapped; bit 0 is now bit 7, bit 1 is
now bit 6, and so on. Now this wouldn't matter much if this convention
was restricted to hardware land. However it seems that at some stage
some of these CRC values were presented at the software level and
someone had to write some code that would interoperate with the
hardware CRC calculation.

In this situation, a normal sane software engineer would simply
reflect each byte before processing it. However, it would seem that
normal sane software engineers were thin on the ground when this early
ground was being broken, because instead of reflecting the bytes,
whoever was responsible held down the byte and reflected the world,
leading to the following "reflected" algorithm which is identical to
the previous one except that everything is reflected except the input
bytes.

```
              Message (non augmented) >-----+
                                            |
           Bytes   0    1    2    3         v
                +----+----+----+----+       |
                |    |    |    |    |>----XOR
                +----+----+----+----+       |
                          ^                 |
                          |                 |
                         XOR                |
                          |                 |
                +----+----+----+----+0      |
                +----+----+----+----+       v
                +----+----+----+----+       |
                +----+----+----+----+       |
                +----+----+----+----+       |
                +----+----+----+----+       |
                +----+----+----+----+       |
                +----+----+----+----+<-----+
                +----+----+----+----+
                +----+----+----+----+
                +----+----+----+----+
                +----+----+----+----+
                +----+----+----+----+255
```

Notes:

   * The table is identical to the one in the previous algorithm
   except that each entry has been reflected.

   * The initial value of the register is the same as in the previous
   algorithm except that it has been reflected.

   * The bytes of the message are processed in the same order as
   before (i.e. the message itself is not reflected).

   * The message bytes themselves don't need to be explicitly
   reflected, because everything else has been!

At the end of execution, the register contains the reflection of the
final CRC value (remainder). Actually, I'm being rather hard on
whoever cooked this up because it seems that hardware implementations
of the CRC algorithm used the reflected checksum value and so
producing a reflected CRC was just right. In fact reflecting the world
was probably a good engineering solution - if a confusing one.

We will call this the REFLECTED algorithm.

Whether or not it made sense at the time, the effect of having
reflected algorithms kicking around the world's FTP sites is that
about half the CRC implementations one runs into are reflected and the
other half not. It's really terribly confusing. In particular, it
would seem to me that the casual reader who runs into a reflected,
table-driven implementation with the bytes "fed in the wrong end"
would have Buckley's chance of ever connecting the code to the concept
of binary mod 2 division.

It couldn't get any more confusing could it? Yes it could.

12. "Reversed" Polys

As if reflected implementations weren't enough, there is another
concept kicking around which makes the situation bizaarly confusing.
The concept is reversed Polys.

It turns out that the reflection of good polys tend to be good polys
too! That is, if G=11101 is a good poly value, then 10111 will be as
well. As a consequence, it seems that every time an organization (such
as CCITT) standardizes on a particularly good poly ("polynomial"),
those in the real world can't leave the poly's reflection alone
either. They just HAVE to use it. As a result, the set of "standard"
poly's has a corresponding set of reflections, which are also in use.
To avoid confusion, we will call these the "reversed" polys.

    X25   standard: 1-0001-0000-0010-0001
    X25   reversed: 1-0000-1000-0001-0001

    CRC16 standard: 1-1000-0000-0000-0101
    CRC16 reversed: 1-0100-0000-0000-0011

Note that here it is the entire poly that is being reflected/reversed,
not just the bottom W bits. This is an important distinction. In the
reflected algorithm described in the previous section, the poly used
in the reflected algorithm was actually identical to that used in the
non-reflected algorithm; all that had happened is that the bytes had
effectively been reflected. As such, all the 16-bit/32-bit numbers in
the algorithm had to be reflected. In contrast, the ENTIRE poly
includes the implicit one bit at the top, and so reversing a poly is
not the same as reflecting its bottom 16 or 32 bits.

The upshot of all this is that a reflected algorithm is not equivalent
to the original algorithm with the poly reflected. Actually, this is
probably less confusing than if they were duals.

If all this seems a bit unclear, don't worry, because we're going to
sort it all out "real soon now". Just one more section to go before
that.

13. Initial and Final Values

In addition to the complexity already seen, CRC algorithms differ from
each other in two other regards:

    * The initial value of the register.

    * The value to be XORed with the final register value.

For example, the "CRC32" algorithm initializes its register to
FFFFFFFF and XORs the final register value with FFFFFFFF.

Most CRC algorithms initialize their register to zero. However, some
initialize it to a non-zero value. In theory (i.e. with no assumptions
about the message), the initial value has no affect on the strength of
the CRC algorithm, the initial value merely providing a fixed starting
point from which the register value can progress. However, in
practice, some messages are more likely than others, and it is wise to
initialize the CRC algorithm register to a value that does not have

"blind spots" that are likely to occur in practice. By "blind spot" is
meant a sequence of message bytes that do not result in the register
changing its value. In particular, any CRC algorithm that initializes
its register to zero will have a blind spot of zero when it starts up
and will be unable to "count" a leading run of zero bytes. As a
leading run of zero bytes is quite common in real messages, it is wise
to initialize the algorithm register to a non-zero value.

14. Defining Algorithms Absolutely

At this point we have covered all the different aspects of
table-driven CRC algorithms. As there are so many variations on these
algorithms, it is worth trying to establish a nomenclature for them.
This section attempts to do that.

We have seen that CRC algorithms vary in:

    * Width of the poly (polynomial).
    * Value of the poly.
    * Initial value for the register.
    * Whether the bits of each byte are reflected before being processed.
    * Whether the algorithm feeds input bytes through the register or
      xors them with a byte from one end and then straight into the table.
    * Whether the final register value should be reversed (as in reflected
      versions).
    * Value to XOR with the final register value.

In order to be able to talk about particular CRC algorithms, we need
to able to define them more precisely than this. For this reason, the
next section attempts to provide a well-defined parameterized model
for CRC algorithms. To refer to a particular algorithm, we need then
simply specify the algorithm in terms of parameters to the model.

15. A Parameterized Model For CRC Algorithms

In this section we define a precise parameterized model CRC algorithm
which, for want of a better name, we will call the "Rocksoft^tm Model
CRC Algorithm" (and why not? Rocksoft^tm could do with some free
advertising :-).

The most important aspect of the model algorithm is that it focusses
exclusively on functionality, ignoring all implementation details. The
aim of the exercise is to construct a way of referring precisely to
particular CRC algorithms, regardless of how confusingly they are
implemented. To this end, the model must be as simple and precise as
possible, with as little confusion as possible.

The Rocksoft^tm Model CRC Algorithm is based essentially on the DIRECT
TABLE ALGORITHM specified earlier. However, the algorithm has to be
further parameterized to enable it to behave in the same way as some
of the messier algorithms out in the real world.

To enable the algorithm to behave like reflected algorithms, we
provide a boolean option to reflect the input bytes, and a boolean
option to specify whether to reflect the output checksum value. By
framing reflection as an input/output transformation, we avoid the
confusion of having to mentally map the parameters of reflected and
non-reflected algorithms.

An extra parameter allows the algorithm's register to be initialized
to a particular value. A further parameter is XORed with the final
value before it is returned.

By putting all these pieces together we end up with the parameters of
the algorithm:

   NAME: This is a name given to the algorithm. A string value.

   WIDTH: This is the width of the algorithm expressed in bits. This
   is one less than the width of the Poly.

   POLY: This parameter is the poly. This is a binary value that
   should be specified as a hexadecimal number. The top bit of the
   poly should be omitted. For example, if the poly is 10110, you
   should specify 06. An important aspect of this parameter is that it
   represents the unreflected poly; the bottom bit of this parameter
   is always the LSB of the divisor during the division regardless of
   whether the algorithm being modelled is reflected.

   INIT: This parameter specifies the initial value of the register
   when the algorithm starts. This is the value that is to be assigned
   to the register in the direct table algorithm. In the table
   algorithm, we may think of the register always commencing with the
   value zero, and this value being XORed into the register after the
   N'th bit iteration. This parameter should be specified as a
   hexadecimal number.

   REFIN: This is a boolean parameter. If it is FALSE, input bytes are
   processed with bit 7 being treated as the most significant bit
   (MSB) and bit 0 being treated as the least significant bit. If this
   parameter is FALSE, each byte is reflected before being processed.

   REFOUT: This is a boolean parameter. If it is set to FALSE, the
   final value in the register is fed into the XOROUT stage directly,
   otherwise, if this parameter is TRUE, the final register value is
   reflected first.

   XOROUT: This is an W-bit value that should be specified as a
   hexadecimal number. It is XORed to the final register value (after
   the REFOUT) stage before the value is returned as the official
   checksum.

   CHECK: This field is not strictly part of the definition, and, in
   the event of an inconsistency between this field and the other
   field, the other fields take precedence. This field is a check
   value that can be used as a weak validator of implementations of
   the algorithm. The field contains the checksum obtained when the
   ASCII string "123456789" is fed through the specified algorithm
   (i.e. 313233... (hexadecimal)).

With these parameters defined, the model can now be used to specify a
particular CRC algorithm exactly. Here is an example specification for
a popular form of the CRC-16 algorithm.

   Name   : "CRC-16"
   Width   : 16
   Poly   : 8005
   Init   : 0000

```
   RefIn  : True
   RefOut : True
   XorOut : 0000
   Check  : BB3D


16. A Catalog of Parameter Sets for Standards
---------------------------------------------
At this point, I would like to give a list of the specifications for
commonly used CRC algorithms. However, most of the algorithms that I
have come into contact with so far are specified in such a vague way
that this has not been possible. What I can provide is a list of polys
for various CRC standards I have heard of:

   X25   standard : 1021        [CRC-CCITT, ADCCP, SDLC/HDLC]
   X25   reversed : 0811

   CRC16 standard : 8005
   CRC16 reversed : 4003        [LHA]

   CRC32          : 04C11DB7    [PKZIP, AUTODIN II, Ethernet, FDDI]

I would be interested in hearing from anyone out there who can tie
down the complete set of model parameters for any of these standards.

However, a program that was kicking around seemed to imply the
following specifications. Can anyone confirm or deny them (or provide
the check values (which I couldn't be bothered coding up and
calculating)).

   Name   : "CRC-16/CITT"
   Width  : 16
   Poly   : 1021
   Init   : FFFF
   RefIn  : False
   RefOut : False
   XorOut : 0000
   Check  : ?


   Name   : "XMODEM"
   Width  : 16
   Poly   : 8408
   Init   : 0000
   RefIn  : True
   RefOut : True
   XorOut : 0000
   Check  : ?


   Name   : "ARC"
   Width  : 16
   Poly   : 8005
   Init   : 0000
   RefIn  : True
   RefOut : True
   XorOut : 0000
   Check  : ?

Here is the specification for the CRC-32 algorithm which is reportedly
```

```
used in PKZip, AUTODIN II, Ethernet, and FDDI.

   Name  : "CRC-32"
   Width : 32
   Poly  : 04C11DB7
   Init  : FFFFFFFF
   RefIn : True
   RefOut : True
   XorOut : FFFFFFFF
   Check : CBF43926
```

17. An Implementation of the Model Algorithm

Here is an implementation of the model algorithm in the C programming
language. The implementation consists of a header file (.h) and an
implementation file (.c). If you're reading this document in a
sequential scroller, you can skip this code by searching for the
string "Roll Your Own".

To ensure that the following code is working, configure it for the
CRC-16 and CRC-32 algorithms given above and ensure that they produce
the specified "check" checksum when fed the test string "123456789"
(see earlier).

```
/****************************************************************************/
/*                       Start of crcmodel.h                            */
/****************************************************************************/
/*                                                                      */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                */
/* Date   : 3 June 1993.                                                */
/* Status : Public domain.                                              */
/*                                                                      */
/* Description : This is the header (.h) file for the reference          */
/* implementation of the Rocksoft^tm Model CRC Algorithm. For more       */
/* information on the Rocksoft^tm Model CRC Algorithm, see the document   */
/* titled "A Painless Guide to CRC Error Detection Algorithms" by Ross    */
/* Williams (ross@guest.adelaide.edu.au.). This document is likely to be in  */
/* "ftp.adelaide.edu.au/pub/rocksoft".                                   */
/*                                                                      */
/* Note: Rocksoft is a trademark of Rocksoft Pty Ltd, Adelaide, Australia.   */
/*                                                                      */
/****************************************************************************/
/*                                                                      */
/* How to Use This Package                                              */
/* -----------------------                                              */
/* Step 1: Declare a variable of type cm_t. Declare another variable     */
/*         (p_cm say) of type p_cm_t and initialize it to point to the first  */
/*         variable (e.g. p_cm_t p_cm = &cm_t).                         */
/*                                                                      */
/* Step 2: Assign values to the parameter fields of the structure.       */
/*         If you don't know what to assign, see the document cited earlier.  */
/*         For example:                                                 */
/*             p_cm->cm_width = 16;                                     */
/*             p_cm->cm_poly  = 0x8005L;                                */
/*             p_cm->cm_init  = 0L;                                     */
/*             p_cm->cm_refin = TRUE;                                   */
/*             p_cm->cm_refot = TRUE;                                   */
/*             p_cm->cm_xorot = 0L;                                     */
/*         Note: Poly is specified without its top bit (18005 becomes 8005).  */
```

```
/*           Note: Width is one bit less than the raw poly width.        */
/*                                                                       */
/* Step 3: Initialize the instance with a call cm_ini(p_cm);             */
/*                                                                       */
/* Step 4: Process zero or more message bytes by placing zero or more    */
/*         successive calls to cm_nxt. Example: cm_nxt(p_cm,ch);         */
/*                                                                       */
/* Step 5: Extract the CRC value at any time by calling crc = cm_crc(p_cm); */
/*         If the CRC is a 16-bit value, it will be in the bottom 16 bits. */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/* Design Notes                                                          */
/* ------------                                                          */
/* PORTABILITY: This package has been coded very conservatively so that  */
/* it will run on as many machines as possible. For example, all external */
/* identifiers have been restricted to 6 characters and all internal ones to */
/* 8 characters. The prefix cm (for Crc Model) is used as an attempt to avoid */
/* namespace collisions. This package is endian independent.             */
/*                                                                       */
/* EFFICIENCY: This package (and its interface) is not designed for      */
/* speed. The purpose of this package is to act as a well-defined reference */
/* model for the specification of CRC algorithms. If you want speed, cook up */
/* a specific table-driven implementation as described in the document cited */
/* above. This package is designed for validation only; if you have found or */
/* implemented a CRC algorithm and wish to describe it as a set of parameters */
/* to the Rocksoft^tm Model CRC Algorithm, your CRC algorithm implementation */
/* should behave identically to this package under those parameters.     */
/*                                                                       */
/*************************************************************************/

/* The following #ifndef encloses this entire */
/* header file, rendering it indempotent.     */
#ifndef CM_DONE
#define CM_DONE


/*************************************************************************/

/* The following definitions are extracted from my style header file which */
/* would be cumbersome to distribute with this package. The DONE_STYLE is the */
/* idempotence symbol used in my style header file.                      */

#ifndef DONE_STYLE

typedef unsigned long    ulong;
typedef unsigned         bool;
typedef unsigned char * p_ubyte_;

#ifndef TRUE
#define FALSE 0
#define TRUE  1
#endif

/* Change to the second definition if you don't have prototypes. */
#define P_(A) A
/* #define P_(A) () */

/* Uncomment this definition if you don't have void. */
/* typedef int void; */
```

```
      #endif

      /***************************************************************************/

      /* CRC Model Abstract Type */
      /* ----------------------- */
      /* The following type stores the context of an executing instance of the  */
      /* model algorithm. Most of the fields are model parameters which must be */
      /* set before the first initializing call to cm_ini.                      */
      typedef struct
        {
        int   cm_width;   /* Parameter: Width in bits [8,32].        */
        ulong cm_poly;    /* Parameter: The algorithm's polynomial. */
        ulong cm_init;    /* Parameter: Initial register value.      */
        bool  cm_refin;   /* Parameter: Reflect input bytes?         */
        bool  cm_refot;   /* Parameter: Reflect output CRC?          */
        ulong cm_xorot;   /* Parameter: XOR this to output CRC.      */

        ulong cm_reg;     /* Context: Context during execution.      */
        } cm_t;
      typedef cm_t *p_cm_t;

      /***************************************************************************/

      /* Functions That Implement The Model */
      /* ---------------------------------- */
      /* The following functions animate the cm_t abstraction. */

      void cm_ini P_((p_cm_t p_cm));
      /* Initializes the argument CRC model instance.          */
      /* All parameter fields must be set before calling this. */

      void cm_nxt P_((p_cm_t p_cm,int ch));
      /* Processes a single message byte [0,255]. */

      void cm_blk P_((p_cm_t p_cm,p_ubyte_ blk_adr,ulong blk_len));
      /* Processes a block of message bytes. */

      ulong cm_crc P_((p_cm_t p_cm));
      /* Returns the CRC value for the message bytes processed so far. */

      /***************************************************************************/

      /* Functions For Table Calculation */
      /* ------------------------------- */
      /* The following function can be used to calculate a CRC lookup table.     */
      /* It can also be used at run-time to create or check static tables.       */

      ulong cm_tab P_((p_cm_t p_cm,int index));
      /* Returns the i'th entry for the lookup table for the specified algorithm.  */
      /* The function examines the fields cm_width, cm_poly, cm_refin, and the     */
      /* argument table index in the range [0,255] and returns the table entry in  */
      /* the bottom cm_width bytes of the return value.                            */

      /***************************************************************************/

      /* End of the header file idempotence #ifndef */
      #endif

      /***************************************************************************/
```

```
/*                           End of crcmodel.h                              */
/****************************************************************************/


/****************************************************************************/
/*                           Start of crcmodel.c                            */
/****************************************************************************/
/*                                                                          */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                    */
/* Date   : 3 June 1993.                                                    */
/* Status : Public domain.                                                  */
/*                                                                          */
/* Description : This is the implementation (.c) file for the reference     */
/* implementation of the Rocksoft^tm Model CRC Algorithm. For more          */
/* information on the Rocksoft^tm Model CRC Algorithm, see the document      */
/* titled "A Painless Guide to CRC Error Detection Algorithms" by Ross       */
/* Williams (ross@guest.adelaide.edu.au.). This document is likely to be in  */
/* "ftp.adelaide.edu.au/pub/rocksoft".                                      */
/*                                                                          */
/* Note: Rocksoft is a trademark of Rocksoft Pty Ltd, Adelaide, Australia.  */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/* Implementation Notes                                                     */
/* --------------------                                                     */
/* To avoid inconsistencies, the specification of each function is not echoed */
/* here. See the header file for a description of these functions.          */
/* This package is light on checking because I want to keep it short and    */
/* simple and portable (i.e. it would be too messy to distribute my entire  */
/* C culture (e.g. assertions package) with this package.                   */
/*                                                                          */
/****************************************************************************/

#include "crcmodel.h"


/****************************************************************************/

/* The following definitions make the code more readable. */

#define BITMASK(X) (1L << (X))
#define MASK32 0xFFFFFFFFL
#define LOCAL static

/****************************************************************************/

LOCAL ulong reflect P_((ulong v,int b));
LOCAL ulong reflect (v,b)
/* Returns the value v with the bottom b [0,32] bits reflected. */
/* Example: reflect(0x3e23L,3) == 0x3e26                        */
ulong v;
int   b;
{
 int   i;
 ulong t = v;
 for (i=0; i<b; i++)
   {
    if (t & 1L)
       v|=  BITMASK((b-1)-i);
     else
       v&= ~BITMASK((b-1)-i);
```

```
    t>>=1;
    }
 return v;
}

/***************************************************************************/

LOCAL ulong widmask P_((p_cm_t));
LOCAL ulong widmask (p_cm)
/* Returns a longword whose value is (2^p_cm->cm_width)-1.       */
/* The trick is to do this portably (e.g. without doing <<32). */
p_cm_t p_cm;
{
 return (((1L<<(p_cm->cm_width-1))-1L)<<1)|1L;
}

/***************************************************************************/

void cm_ini (p_cm)
p_cm_t p_cm;
{
 p_cm->cm_reg = p_cm->cm_init;
}

/***************************************************************************/

void cm_nxt (p_cm,ch)
p_cm_t p_cm;
int    ch;
{
 int   i;
 ulong uch  = (ulong) ch;
 ulong topbit = BITMASK(p_cm->cm_width-1);

 if (p_cm->cm_refin) uch = reflect(uch,8);
 p_cm->cm_reg ^= (uch << (p_cm->cm_width-8));
 for (i=0; i<8; i++)
    {
     if (p_cm->cm_reg & topbit)
        p_cm->cm_reg = (p_cm->cm_reg << 1) ^ p_cm->cm_poly;
     else
        p_cm->cm_reg <<= 1;
     p_cm->cm_reg &= widmask(p_cm);
    }
}

/***************************************************************************/

void cm_blk (p_cm,blk_adr,blk_len)
p_cm_t   p_cm;
p_ubyte_ blk_adr;
ulong    blk_len;
{
 while (blk_len--) cm_nxt(p_cm,*blk_adr++);
}

/***************************************************************************/

ulong cm_crc (p_cm)
p_cm_t p_cm;
```

```
{
 if (p_cm->cm_refot)
    return p_cm->cm_xorot ^ reflect(p_cm->cm_reg,p_cm->cm_width);
 else
    return p_cm->cm_xorot ^ p_cm->cm_reg;
}

/****************************************************************************/

ulong cm_tab (p_cm,index)
p_cm_t p_cm;
int    index;
{
 int   i;
 ulong r;
 ulong topbit = BITMASK(p_cm->cm_width-1);
 ulong inbyte = (ulong) index;

 if (p_cm->cm_refin) inbyte = reflect(inbyte,8);
 r = inbyte << (p_cm->cm_width-8);
 for (i=0; i<8; i++)
    if (r & topbit)
       r = (r << 1) ^ p_cm->cm_poly;
    else
       r<<=1;
 if (p_cm->cm_refin) r = reflect(r,p_cm->cm_width);
 return r & widmask(p_cm);
}

/****************************************************************************/
/*                              End of crcmodel.c                           */
/****************************************************************************/
```

18. Roll Your Own Table-Driven Implementation

Despite all the fuss I've made about understanding and defining CRC
algorithms, the mechanics of their high-speed implementation remains
trivial. There are really only two forms: normal and reflected. Normal
shifts to the left and covers the case of algorithms with Refin=FALSE
and Refot=FALSE. Reflected shifts to the right and covers algorithms
with both those parameters true. (If you want one parameter true and
the other false, you'll have to figure it out for yourself!) The
polynomial is embedded in the lookup table (to be discussed). The
other parameters, Init and XorOt can be coded as macros. Here is the
32-bit normal form (the 16-bit form is similar).

```
   unsigned long crc_normal ();
   unsigned long crc_normal (blk_adr,blk_len)
   unsigned char *blk_adr;
   unsigned long  blk_len;
   {
    unsigned long crc = INIT;
    while (blk_len--)
       crc = crctable[((crc>>24) ^ *blk_adr++) & 0xFFL] ^ (crc << 8);
    return crc ^ XOROT;
   }
```

Here is the reflected form:

```
   unsigned long crc_reflected ();
   unsigned long crc_reflected (blk_adr,blk_len)
   unsigned char *blk_adr;
   unsigned long  blk_len;
   {
    unsigned long crc = INIT_REFLECTED;
    while (blk_len--)
       crc = crctable[(crc ^ *blk_adr++) & 0xFFL] ^ (crc >> 8));
    return crc ^ XOROT;
   }
```

Note: I have carefully checked the above two code fragments, but I
haven't actually compiled or tested them. This shouldn't matter to
you, as, no matter WHAT you code, you will always be able to tell if
you have got it right by running whatever you have created against the
reference model given earlier. The code fragments above are really
just a rough guide. The reference model is the definitive guide.

Note: If you don't care much about speed, just use the reference model
code!

19. Generating A Lookup Table

The only component missing from the normal and reversed code fragments
in the previous section is the lookup table. The lookup table can be
computed at run time using the cm_tab function of the model package
given earlier, or can be pre-computed and inserted into the C program.
In either case, it should be noted that the lookup table depends only
on the POLY and RefIn (and RefOt) parameters. Basically, the
polynomial determines the table, but you can generate a reflected
table too if you want to use the reflected form above.

The following program generates any desired 16-bit or 32-bit lookup
table. Skip to the word "Summary" if you want to skip over this code.

/**/
/*   Start of crctable.c   */
/**/
/*   */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).   */
/* Date   : 3 June 1993.   */
/* Version : 1.0.   */
/* Status : Public domain.   */
/*   */
/* Description : This program writes a CRC lookup table (suitable for   */
/* inclusion in a C program) to a designated output file. The program can be */
/* statically configured to produce any table covered by the Rocksoft^tm   */
/* Model CRC Algorithm. For more information on the Rocksoft^tm Model CRC */
/* Algorithm, see the document titled "A Painless Guide to CRC Error   */
/* Detection Algorithms" by Ross Williams (ross@guest.adelaide.edu.au.). This */
/* document is likely to be in "ftp.adelaide.edu.au/pub/rocksoft".   */
/*   */
/* Note: Rocksoft is a trademark of Rocksoft Pty Ltd, Adelaide, Australia. */
/*   */
/**/

#include <stdio.h>
```

```
#include <stdlib.h>
#include "crcmodel.h"

/***************************************************************************/

/* TABLE PARAMETERS                                                     */
/* ===============                                                      */
/* The following parameters entirely determine the table to be generated. You */
/* should need to modify only the definitions in this section before running  */
/* this program.                                                        */
/*                                                                      */
/*    TB_FILE  is the name of the output file.                          */
/*    TB_WIDTH is the table width in bytes (either 2 or 4).             */
/*    TB_POLY  is the "polynomial", which must be TB_WIDTH bytes wide.  */
/*    TB_REVER indicates whether the table is to be reversed (reflected).   */
/*                                                                      */
/* Example:                                                             */
/*                                                                      */
/*    #define TB_FILE   "crctable.out"                                  */
/*    #define TB_WIDTH  2                                               */
/*    #define TB_POLY   0x8005L                                         */
/*    #define TB_REVER  TRUE                                            */

#define TB_FILE   "crctable.out"
#define TB_WIDTH  4
#define TB_POLY   0x04C11DB7L
#define TB_REVER  TRUE

/***************************************************************************/

/* Miscellaneous definitions. */

#define LOCAL static
FILE *outfile;
#define WR(X) fprintf(outfile,(X))
#define WP(X,Y) fprintf(outfile,(X),(Y))

/***************************************************************************/

LOCAL void chk_err P_((char *));
LOCAL void chk_err (mess)
/* If mess is non-empty, write it out and abort. Otherwise, check the error  */
/* status of outfile and abort if an error has occurred.                     */
char *mess;
{
 if (mess[0] != 0  ) {printf("%s\n",mess); exit(EXIT_FAILURE);}
 if (ferror(outfile)) {perror("chk_err");   exit(EXIT_FAILURE);}
}

/***************************************************************************/

LOCAL void chkparam P_((void));
LOCAL void chkparam ()
{
 if ((TB_WIDTH != 2) && (TB_WIDTH != 4))
    chk_err("chkparam: Width parameter is illegal.");
 if ((TB_WIDTH == 2) && (TB_POLY & 0xFFFF0000L))
    chk_err("chkparam: Poly parameter is too wide.");
 if ((TB_REVER != FALSE) && (TB_REVER != TRUE))
    chk_err("chkparam: Reverse parameter is not boolean.");
```

```
}

/***************************************************************************/

LOCAL void gentable P_((void));
LOCAL void gentable ()
{
 WR("/***************************************************************/\n");
 WR("/*                                                             */\n");
 WR("/* CRC LOOKUP TABLE                                            */\n");
 WR("/* ================                                            */\n");
 WR("/* The following CRC lookup table was generated automagically  */\n");
 WR("/* by the Rocksoft^tm Model CRC Algorithm Table Generation     */\n");
 WR("/* Program V1.0 using the following model parameters:          */\n");
 WR("/*                                                             */\n");
 WP("/*    Width  : %1lu bytes.                                      */\n",
    (ulong) TB_WIDTH);
 if (TB_WIDTH == 2)
 WP("/*    Poly   : 0x%04lX                                         */\n",
    (ulong) TB_POLY);
 else
 WP("/*    Poly   : 0x%08lXL                                        */\n",
    (ulong) TB_POLY);
 if (TB_REVER)
 WR("/*    Reverse : TRUE.                                          */\n");
 else
 WR("/*    Reverse : FALSE.                                         */\n");
 WR("/*                                                             */\n");
 WR("/* For more information on the Rocksoft^tm Model CRC Algorithm, */\n");
 WR("/* see the document titled \"A Painless Guide to CRC Error      */\n");
 WR("/* Detection Algorithms\" by Ross Williams                      */\n");
 WR("/* (ross@guest.adelaide.edu.au.). This document is likely to be */\n");
 WR("/* in the FTP archive \"ftp.adelaide.edu.au/pub/rocksoft\".      */\n");
 WR("/*                                                             */\n");
 WR("/***************************************************************/\n");
 WR("\n");
 switch (TB_WIDTH)
   {
    case 2: WR("unsigned short crctable[256] =\n{\n"); break;
    case 4: WR("unsigned long  crctable[256] =\n{\n"); break;
    default: chk_err("gentable: TB_WIDTH is invalid.");
   }
 chk_err("");

 {
  int i;
  cm_t cm;
  char *form   = (TB_WIDTH==2) ? "0x%04lX" : "0x%08lXL";
  int   perline = (TB_WIDTH==2) ? 8 : 4;

  cm.cm_width = TB_WIDTH*8;
  cm.cm_poly  = TB_POLY;
  cm.cm_refin = TB_REVER;

  for (i=0; i<256; i++)
    {
     WR(" ");
     WP(form,(ulong) cm_tab(&cm,i));
     if (i != 255) WR(",");
     if (((i+1) % perline) == 0) WR("\n");
```

```
    chk_err("");
    }

 WR("};\n");
 WR("\n");
 WR("/****************************************************************/\n");
 WR("/*                    End of CRC Lookup Table                   */\n");
 WR("/****************************************************************/\n");
 WR("");
 chk_err("");
}
}

/*****************************************************************************/

main ()
{
 printf("\n");
 printf("Rocksoft^tm Model CRC Algorithm Table Generation Program V1.0\n");
 printf("----------------------------------------------------------------\n");
 printf("Output file is \"%s\".\n",TB_FILE);
 chkparam();
 outfile = fopen(TB_FILE,"w"); chk_err("");
 gentable();
 if (fclose(outfile) != 0)
    chk_err("main: Couldn't close output file.");
 printf("\nSUCCESS: The table has been successfully written.\n");
}

/*****************************************************************************/
/*                          End of crctable.c                                */
/*****************************************************************************/
```

20. Summary

This document has provided a detailed explanation of CRC algorithms
explaining their theory and stepping through increasingly
sophisticated implementations ranging from simple bit shifting through
to byte-at-a-time table-driven implementations. The various
implementations of different CRC algorithms that make them confusing
to deal with have been explained. A parameterized model algorithm has
been described that can be used to precisely define a particular CRC
algorithm, and a reference implementation provided. Finally, a program
to generate CRC tables has been provided.

21. Corrections

If you think that any part of this document is unclear or incorrect,
or have any other information, or suggestions on how this document
could be improved, please context the author. In particular, I would
like to hear from anyone who can provide Rocksoft^tm Model CRC
Algorithm parameters for standard algorithms out there.

A. Glossary

CHECKSUM - A number that has been calculated as a function of some
message. The literal interpretation of this word "Check-Sum" indicates
that the function should involve simply adding up the bytes in the
message. Perhaps this was what early checksums were. Today, however,
although more sophisticated formulae are used, the term "checksum" is

still used.

CRC - This stands for "Cyclic Redundancy Code". Whereas the term "checksum" seems to be used to refer to any non-cryptographic checking information unit, the term "CRC" seems to be reserved only for algorithms that are based on the "polynomial" division idea.

G - This symbol is used in this document to represent the Poly.

MESSAGE - The input data being checksummed. This is usually structured as a sequence of bytes. Whether the top bit or the bottom bit of each byte is treated as the most significant or least significant is a parameter of CRC algorithms.

POLY - This is my friendly term for the polynomial of a CRC.

POLYNOMIAL - The "polynomial" of a CRC algorithm is simply the divisor in the division implementing the CRC algorithm.

REFLECT - A binary number is reflected by swapping all of its bits around the central point. For example, 1101 is the reflection of 1011.

ROCKSOFT^TM MODEL CRC ALGORITHM - A parameterized algorithm whose purpose is to act as a solid reference for describing CRC algorithms. Typically CRC algorithms are specified by quoting a polynomial. However, in order to construct a precise implementation, one also needs to know initialization values and so on.

WIDTH - The width of a CRC algorithm is the width of its polynomical minus one. For example, if the polynomial is 11010, the width would be 4 bits. The width is usually set to be a multiple of 8 bits.

B. References

[Griffiths87] Griffiths, G., Carlyle Stones, G., "The Tea-Leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32", Communications of the ACM, 30(7), pp.617-620. Comment: This paper describes a high-speed table-driven implementation of CRC algorithms. The technique seems to be a touch messy, and is superceded by the Sarwete algorithm.

[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.

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.

Ramabadran T.V., Gaitonde S.S., "A tutorial on CRC computations", IEEE
Micro, Aug 1988.

Schwaderer W.D., "CRC Calculation", April 85 PC Tech Journal,
pp.118-133.

Ward R.K, Tabandeh M., "Error Correction and Detection, A Geometric
Approach" The Computer Journal, Vol. 27, No. 3, 1984, pp.246-253.

Wecker, S., "A Table-Lookup Algorithm for Software Computation of
Cyclic Redundancy Check (CRC)," Digital Equipment Corporation
memorandum, 1974.

--<End of Document>--

<icdlj@asuacad.bitnet>   Dave Janecek -Electronic Field Service
<icdlj@asuvm.inre.asu.edu>   Tech. c/o I.R.M. Tech Shop-0201, ASU,
                                         Tempe, Arizona 85287-0201 :602-965-9126
                                                      or home :602-832-7127