Whirlpool

CRC calculations: the whys and the wherefores

by Julian Bucknall

I was debugging a subtle bug in a test program the other day, one that used our Abbrevia product. Nothing seemed to make sense: the file was being decrypted properly (or so it seemed), yet the deflate routine was returning an invalid CRC (*Cyclic Redundancy Check*) error. The bizarre thing was the deflated file was the correct size, and the text looked to be correct. I resorted to adding debugging statements to try and track it down. For every character unpacked for the archive, I added code to write out the character and an updated CRC value including that character. I then unpacked the file with WinZip and wrote a little application that calculated the CRC for the unpacked file, character by character. I then compared both debug logs and found that Abbrevia was unpacking a single character in the middle of the file incorrectly (it involved the number 32768, believe it or not). From there, it was the work of moments to find the bug and fix it.

That little tale wasn't to brag about my debugging skills, or to admit that Abbrevia had that particular bug, but it got me thinking. Abbrevia was calculating the CRC by using this big table of 256 `longints`. How had we, TurboPower, originally built this table? Did we copy it from some other source, way back when, or had we originally written a program to generate it? I decided to investigate, and this article is what I discovered. I warn you now: there is mathematics ahead (the mention of which is enough to give Our Esteemed Editor the screaming heebie-jeebies, not because he won't understand it, but because he has to convert my Word document with italic *x* and ^{super}scripts all over the place into Ventura Publisher). Also ahead are some downright bizarre bit manipulations.

Bring It On

CRCs are a form of checksum. A *checksum* is some kind of arithmetic operation on a block of data (usually known as the *message*) to produce an integer value. This integer value can then be used to check that the block hasn't changed through being transmitted over a phone line between modems, or after having been stored on some storage medium or in some archive. The theory goes like this: if the checksum is stored with the block of data, when the data is received, the reader application can recalculate the checksum for the block and compare it with the stored checksum. There are two possibilities for the comparison. First, the computed and the stored checksum do not match, in which case either the block was not read properly or was corrupted, or that the stored checksum itself was not read properly or was corrupted. We can't tell either way so we assume that the block was corrupted and we try and read it again. The second possibility is that the computed and stored checksums match, in which case we assume that the block of data is valid.

Of course, in the second case the block of data could still be invalid but, luckily (or unluckily, depending on your point of view), happens to generate the same checksum as the original. The theory of checksums aims to improve the chances of detecting errors in the received block of data.

There are three main uses of checksums. The first is the obvious one: detecting transmission errors when sending a block of data across some phone line which is prone to noise (it is assumed that the noise would cause bit errors in the data being transmitted: a 0 gets received as a 1, and so on). The second use of checksums is by disk controllers to verify that the data read from a sector equals the data that was originally written there. Compression programs use CRCs to verify that the data compressed in an archive decompresses properly (in other words that the archive file was not corrupted, or that the compressor or decompressor code did not have bugs, as in my debugging session).

In the early days, for example with the early XMODEM file transfer protocol, the checksum was a simple sum of all the bytes in a transmission block, modulo 256 (the simple way to calculate this is to add up all the byte values in the block using a byte variable; the implicit discarding of the overflow is equivalent to taking the sum mod 256). There is one big problem with this checksum: it's not very good at detecting errors in blocks over 256 bytes in size (which is one reason why the XMODEM protocol uses 128 byte blocks). If two bytes get swapped through transmission errors, the checksum from the bad block will equal the checksum from the good block. If two bits get flipped in one byte and the same two bits get flipped in another: this simple checksum wouldn't notice.

A better checksum in this case would be a hash algorithm, such as those I discussed in my two-parter on hash tables (*The Delphi Magazine*, February and March 1998). Different bytes in the block affect the hash value in different ways, so simple transpositions of bits and bytes are not missed as easily. Hash values are usually 32-bit integers, meaning that, providing the hash function was a

good one, the probability of two blocks of data having the same hash value is one in 2^{32}.

Deep Water

CRCs, however, provide a better checksum than, say, ELF hash (and can indeed be used as hash values). The reason is that CRCs are calculated with very simple bit shifts and XOR operations, and hence are very easy to program in hardware. The mathematics behind them (Galois Field theory) enable us to accurately predict the effects of various types of error. In fact, the standard CRC-16 algorithm has been devised to detect all errors that change an odd number of bits, all errors that change 2 bits (providing the block is less than 32,767 bits long), all errors that consist of a single burst of 16 or fewer bits, and so on.

So what exactly are these CRC values, and just how is a CRC calculated? A CRC is a checksum on the entire block of data, but instead of using addition as the operation on each byte, it uses division. A CRC is the remainder of a division of the block of data (being viewed as a very large polynomial) by another, smaller, polynomial, using modulo 2 arithmetic.

Before you start slowly shuffling away, whistling nonchalantly, wondering why I haven't been locked up yet, let's go back to high school *[secondary school, Julian: we know you're a Brit! Ed]* and do some long division. Way back when, before calculators, we learned how to do division longhand. An example is shown in Figure 1, where we divide 34,567 (the *dividend*) by 12 (the *divisor*) to give 2,880 (the *quotient*) with 7 left over (the *remainder*).

➤ *Figure 1: Long division.*

```
        2 8 8 0
    -----------
1 2 ) 3 4 5 6 7
      2 4
      ---
      1 0 5
        9 6
      -----
          9 6
          9 6
          ---
            0 7
              0
            ---
              7
```

```
                3x^3 - 2x^2 + 9x -12
          ------------------------
x + 2 ) 3x^4 + 4x^3 + 5x^2 + 6x + 7
        3x^4 + 6x^3
        --------
             -2x^3 + 5x^2
             -2x^3 - 4x^2
             ---------
                    9x^2 + 6x
                    9x^2 +18x
                    --------
                         -12x + 7
                         -12x -24
                         --------
                               31
```

➤ *Figure 2: Polynomial division.*

Simple, huh? But I wonder how many people learned how to do polynomial division: division of one polynomial by another? Well, it's just like long division, really. Let's divide $3x^4 + 4x^3 + 5x^2 + 6x + 7$ by $x + 2$. A nice simple example to set us off! Well, we just proceed as for long division, except that the coefficients at each stage do not carry over into the next column, as we do with base 10 division. $x + 2$ divides into $3x^4 + 4x^3$, $3x^3$ times with remainder $-2x^3$. We start building up the polynomial division the same way we did long division; follow along with Figure 2.

Bring down the $5x^2$ and continue. $x + 2$ divides into $-2x^3 + 5x^2$, $-2x^2$ times, with remainder $9x^2$. Bring down the $6x$. $x + 2$ divides into $9x^2 + 6x$, $9x$ times, with remainder $-12x$. Finally bring down the 7. $x + 2$ divides into $-12x + 7$, -12 times, with remainder 31. So that's the answer: $3x^4 + 4x^3 + 5x^2 + 6x + 7$ divided by $x + 2$ gives $3x^3 - 2x^2 + 9x - 12$, with remainder 31.

Notice that the degree of the remainder polynomial is less than the degree of the divisor polynomial. (The *degree* of a polynomial is the largest power of x in the polynomial: the degree of the dividend in our example is 4, of the divisor, 1, of the remainder, 0.)

Notice something else, though. We don't need those x, x^2 terms and the like, they can just be removed and their presence assumed. Figure 3 shows the same division with no x terms; we are dividing [3 4 5 6 7] by [1 2] to give [3 -2 9 -12] with remainder [31], this representation saving both me and OEE some typographic effort.

It looks a little more like long division now, doesn't it? Again the main difference is that there are no carries between coefficients, which is why you see those weird terms with minus signs.

Fast Changes

Now that you can see how polynomial division works, let's now imagine that the polynomials we want to use as dividends and divisors have coefficients that are either 0 or 1 only, *binary polynomials* if you like. Figure 4 shows the division of [1 0 1 1 1 0] by [1 0 1] to give [1 0 0 1] with remainder [1 -1]. In fact, the coefficients all turn out to be 0 or 1 in the division sum as well, except for that pesky -1 that managed to creep in to spoil the party. How to get rid of it?

Enter modulo 2 arithmetic. When I was at school (it's certainly Nostalgia City around here this month) we learnt about modulo 12

```
          3  -2   9 -12
      ------------------
1 2 ) 3   4   5   6   7
      3   6
      -----
         -2   5
         -2  -4
         ------
              9   6
              9 +18
              -----
                -12   7
                -12 -24
                -------
                     31
```

➤ *Figure 3: Polynomial division with no x in sight.*

arithmetic as *clock arithmetic*. We pretended that the 12 on a clock face was 0, and then used the face to do arithmetic. 5 + 9 equals 2 in clock arithmetic (ie advance 9 hours from 5 o'clock to make 2 o'clock), and so 2 - 9 equals 5. No negatives! At university, of course, we extended this to different bases and learnt chunky theorems by Fermat about it, but for now I want you to think about what modulo 2 arithmetic means.

Well, for a kick-off, the only digits we have to worry about are 0 and 1. Let's consider addition: 0 + 0 equals 0. 0 + 1 equals 1, as does 1 + 0. And 1 + 1 equals 0. (If you have trouble understanding this result, consider a clock face with just two points, 0 and 1. Adding 1 to 1 would bring you back to 0 again.) And

subtraction? 0 - 0 = 0; 0 - 1 = 1; 1 - 0 = 1; and 1 - 1 = 0.

But wait! Have a look at that again. Addition and subtraction are *equivalent* in modulo 2 arithmetic, it doesn't matter whether the operator is a plus or a minus, the answer is the same. Those of you who are boolean buffs will also notice something else: addition or subtraction modulo 2 is equivalent to the XOR operation: 0 XOR 0 = 0, 1 XOR 0 = 1, and so on. Also, in moludo 2 arithmetic, multiply is equivalent to AND; the proof is left as a very easy exercise for the reader.

If we use modulo 2 arithmetic with the binary polynomial division in Figure 4, we end up with the remainder of [1 1]. All coefficients are now 0 or 1; all operations turn out to be XOR operations. Suddenly we are in bit twiddling land and we can leave the explicit polynomials in *x* behind; we'll still refer to the components as polynomials though to emphasize the fact we're using modulo 2 arithmetic.

Notice that if the divisor polynomial has *n* bits (the degree is *n*-1), then the remainder is at most *n*-1 bits in length. This is the equivalent of saying in normal long division that the remainder is always less than the divisor.

So this is CRC, then. We take a block of data and view it as a very, very long binary polynomial, with its coefficients being the individual bits in the block. We then divide this very long binary polynomial using modulo 2 arithmetic with a special magic polynomial (usually called the *generating polynomial*) and the remainder of this division is the CRC value.

The reason I'm going to call the generating polynomial the magic polynomial in this article is that the mathematics to identify good generating polynomials from bad ones is waaaaay beyond normal school maths and the majority of my readers (which is Dilbertspeak for 'I've never taken a course on Galois Field theory, so I don't know the maths either').

A couple of examples of magic polynomials are, [1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1] for CRC-16 (which can be written as $18005), and [1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1] for CRC-CCITT (which can be written as ($11021). Both of these are 17-bit polynomials and hence would generate 16-bit CRC values. (In fact, as it happens, it turns out that the reversed polynomials are also good magic polynomials.) For CRC-32, the divisor is a 33-bit polynomial with 1s at positions 32, 26, 23, 22, 16, 12, 11, 10, 8, 7, 5, 4, 2, 1, and 0, and would generate 32-bit CRCs (this polynomial can be written as $104C11DB7).

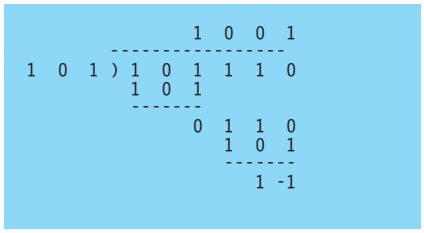
So, is that the end of the article then? Well, no, not unless you want to write a Delphi routine for dividing an 8,192-bit polynomial by a 17-bit polynomial to calculate the CRC for a 1,024-byte block with CRC-16. I certainly don't, so we'll investigate ways of making the calculation easier.

Latest Craze

Think back to our long division example where we were dividing 34,567 by 12. We started off by selecting the leftmost two digits, 34, and then dividing them by 12. At that point, we didn't really know or care what the following digits of the dividend would have been, we could have been dividing 34 by 12 or 34,456,789 by 12: it didn't matter. We only cared about the remaining terms in the dividend when we had to bring the next digit down for the next stage of the division.

So let's devise an algorithm for calculating the remainder of dividing a 1,000-digit number (or whatever) by 123, say. Notice we don't care really how big the dividend is, we just know it's larger than `MaxLongInt` and needs a flipping long string to hold the digits. Remember we don't want the quotient, we are only interested in the remainder. (If you look at Figure 1 again, you may see what's going on better.)

Assume we have a work variable to hold the current remainder. This can be a simple integer variable. In CRC-land this variable is known as the *register*. Initialize it to 0; in other words, we start off with a remainder of 0. We shall read the



➤ *Figure 4: Binary polynomial division.*

dividend, digit by digit from the left.

1. Get the next digit from the dividend. If there is no next digit (ie we've exhausted the dividend string), stop. The register holds the remainder.

2. Otherwise, multiply the register by 10 and add this next digit from the dividend.

3. Calculate the register value mod 123, and put it back into the register. Go back to step 1.

That's it. Pretty simple, huh? We didn't need to know how big the dividend was, we just concentrated on each individual digit, performed a couple of simple arithmetic operations on each, and, before we knew what was happening, we'd calculated the remainder.

So, this is what we do with binary polynomials to calculate the CRC. It's made even easier because we don't have to multiply or apply the mod operator, we just shift bits and XOR. Let's outline the algorithm for a 17-bit divisor (eg CRC-16), we'll store it in a longint. We need a register variable again; we'll make it 32-bits in size, a longint, so we can manipulate it with the 17-bit divisor. We initialize it to 0 as before. We assume that the block can be read bit by bit (we'll get the bits in a byte from the most significant bit down to the least significant bit). I shall assume that the least significant bit of the register is known as bit 0, and the most significant bit is known as bit 31. Bit 16 is then the start of the second word of the `longint` register.

Take a look at Figure 4 to see what's going on as I show the algorithm.

1. Get the next bit from the block. If there is no next bit (ie

we've exhausted the block), stop. The lower 16 bits of the register hold the remainder.

2. Shift the register left by one bit and add the next bit from the block as bit zero.

3. If the bit at position 16 of the register is 0, go back to step 1.

4. If the bit at position 16 is 1, XOR the magic polynomial with the register.

This looks a little funny compared with the normal arithmetic one. What's all this testing of the bit at position 16 of the register for? Well, in modulo 2 arithmetic, we know that a polynomial will only divide another if the dividend is at least the same degree as the divisor. The divisor is of degree 16, so we test bit 16 of the dividend to be 1. Once it is we know we can divide the register by the divisor (it'll always divide once) and then we subtract (ie XOR) one times the divisor from the register to give the new remainder. Shifting left by one is of course equivalent to multiplying by 2. So, you see, it's pretty well equivalent to normal arithmetic.

But, notice one other thing. We never really use the top 16 bits of the register, nor of the divisor. We can get away by assuming that bit 16 of the divisor is always 1 and, when we perform the XOR, bit 16 of the register always gets set to zero. So, here's the better CRC algorithm with register and divisor both 16 bits long.

1. Get the next bit from the block. If there is no next bit (ie we've exhausted the block), stop. The register holds the remainder.

2. If the top bit of the register is 0, shift the register left by one bit, and add the next bit from the block as bit zero. Go to step 1.

3. Otherwise, shift the register left by one bit, add the next bit from the block as bit zero, XOR the magic polynomial with the register. Go to step 1.

Time for some Delphi code, methinks. Listing 1 has a standard, pretty slow, implementation of the 16-bit CRC as used by XMODEM/CRC. The first parameter is the buffer, and the second the number of bytes in the buffer. The third parameter is `aPoly`. This is the magic polynomial we'll be using for divisor, less the most significant bit. For XMODEM/CRC the value of `aPoly` is $1021. Another polynomial you can use if you want to experiment is the one for the CRC-16 algorithm: $8005. The return value type is defined as `TaaCRC16` (a word) and will be the resulting CRC.

The routine with the $1021 polynomial gives the same answers as the standard table-driven XMODEM/CRC calculation, so we're obviously on the right track.

➤ *Listing 1: Simple CRC calculation (XMODEM/CRC).*

```
function AAGet16BitCRCStd(var aBuffer; aBufLen : integer;
  aMagicPoly : TaaCRC16) : TaaCRC16;
const
  TopmostBitMask = $8000;
var
  i   : integer;
  Buf : TByteArray absolute aBuffer;
  Reg : TaaCRC16;
  B   : byte;
  bit : integer;
begin
  {initialise the register}
  Reg := 0;
  {do for all bytes in the buffer...}
  for i := 0 to pred(aBufLen) do begin
    B := Buf[i];
    {do for all bits in the current byte}
    for bit := 0 to 7 do begin
      {if the high bit of the register is 1, shift the register left by one, xor
       in the next bit from the byte, and xor the magic polynomial}
      if ((Reg and TopmostBitMask) <> 0) then
        Reg := (Reg shl 1) xor (B shr 7) xor aMagicPoly
      {otherwise the high bit of the register is 0, shift the register
       left by one, xor in the next bit from the byte}
      else
        Reg := (Reg shl 1) xor (B shr 7);
      B := B shl 1;
    end;
  end;
  {return the register}
  Result := Reg;
end;
```

Dreaming In Metaphors

Note that I said Listing 1 was slow. The reason it's slow is that we are processing the block bit by bit. It would be interesting to see if there was a way to process the block a byte at a time instead.

Consider the case where we're halfway through calculating the CRC for a block of data. We've just read a new byte from the block, and we're about to feed it into the register. Suppose the top byte of the register has bits labelled r7, r6, r5, r4, r3, r2, r1, and r0. r7 then decides for us whether we are about to XOR the magic polynomial into the register. If r7 is clear we won't XOR, if it is set we will. Let the top byte of the magic polynomial be p7, p6, ..., p1, p0.

After the first bit of the input byte is fed into the register, the top part of the polynomial either looks like this:

```
r6 r5 r4 r3 r2 r1 r0 xx
```

if r7 was clear (where xx was the top bit of the least significant byte of the register), or like this:

```
(r6 + p7) ... (r0 + p1)
  (xx + p0)
```

if r7 was set (+ is equivalent to XOR, of course). We could represent these two cases as one:

```
(r6 + r7*p7) ... (r0 + r7*p1)
  (xx + r7*p0)
```

where * is the usual multiplication operator. (This isn't immediately obvious, at least it wasn't to me, but if you draw up a table of the possible bit values, you'll see it is so.) The new top bit of the register thus depends on the values of both r6 and r7.

Feed in the next bit. It's going to get a little more complicated now, so be warned! Again we have two cases: either (r6 + r7*p7) is 0 or it is 1. If you do the math (it's made easier since $x + x = 0$ and $x * x = x$ in modulo 2 arithmetic), here is the value of the new top bit in the top byte of the register:

```
r5 + r7*p6 + (r6+r7)*p7
```

Brrr. But wait a moment. This says that after two bits being fed into the register and the shifts and XORs having been done, the top bit is a direct calculation on bits r7, r6 and r5 of the original register. If we do it 6 more times, we get this as the value in the top bit:

```
xx + r7*p0 + r6*p1 + r5*p2 + r4*p3
   + r3*p4 + r2*p5 + r1*p6
   + (r0+r1+r2+r3+r4+r5+r6+r7)*p7
```

or, to put it in layman's terms:

```
xx + (a mess with rs and ps)
```

(where xx was the top bit of the original lower byte). Other bits in the register after eight applications of this shift-and-XOR algorithm have the same form: they consist of an original bit shifted left 8 times, XORed with a whole bunch of terms that depend on the bits in the original top byte of the register and the magic polynomial. What does all this tell us (apart from that I have entirely too much time on my hands to work this lot out)? Firstly, the top byte of the original register has gone, it's been shifted out (duh!). Of course, it leaves behind some 'memory' of its passing in that mess of algebra, we'll come back to that in a moment. Secondly, we've shifted in a byte from the block of data into the least significant byte of the register (the original least significant byte has been shifted up of course). Finally, during all these shifts and whatnot, the register as a whole was subjected to a series of XORs, the number and timing of which depended solely on the bits in the top register byte.

We are religiously feeding in bits from the message into the bottom of the register, and they are slowly making their way up to the top, at which point they play a part in the determination of whether we XORed the magic polynomial or not. As the bits moved up the register they played no part in the decision to XOR or not. In fact, they might just well not have been there.

So, instead of doing the XORs of the magic polynomial piecemeal, one by one, as we shift and compare the top bit, we could 'add' them up eight at a time into one super magic value (well, I call it that because it does come from the magic polynomial) which we could then XOR after we'd shifted the register by eight bits, ie a byte, and moved in a byte from the block of data.

The super magic value depends on the magic polynomial of course, but also on the value of the top byte of the register (that's what the algebra told us). Since there are only 256 different values of the top byte of the register, we could create a table of 256 16-byte super magic values at initialization-time (or at compile-time) for our particular magic polynomial. The CRC algorithm then becomes:

1. Get the next byte from the block of data. If there are none left, stop; the register contains the CRC value.
2. Get the top byte of the register.
3. Shift the register left by a byte, add in the next byte from the data block.
4. XOR the register with the super magic value from the table, indexed by the original top byte. Return to step 1.

And that's all (!) there is to it. The algorithm explains the big table found in Abbrevia. Unfortunately it *doesn't* explain how the table is generated, so we've a little more to do.

People Asking Why

What we do to generate the table is this. The algebra we waded through showed us that the super magic values in the table depend only on the top byte of the register. What we'd like to do is to write a routine to perform the standard CRC shift-and-XOR algorithm on a register whose top byte varies from 0 to 256. The value in the register after applying the algorithm is the super magic value to go into the table. Great idea, but what about the other byte of the register? What about the byte that gets fed in?

Well, we showed that each bit in the final register either gets shifted out, or is the bit originally from 8 places to the right, XORed with some expression involving the top bits of the original register and the magic polynomial. If the original register was r7, r6, ..., r0, s7, s6, ..., s0 (ie 16 bits) and the byte that was moved in was b7, b6, ..., b0, then the final value of the register after 8 bit cycles is: s7+SMV15, s6+SMV14, ..., s0+SMV8, b7+SMV7, b6+SMV6, ..., b0+SMV0, where the SMVs are the corresponding bits of the super magic value. Easy enough: make the s bits and the b bit all zero, ie the lower byte of the register and the byte that's fed in both zero. The value of the register afterwards is the super magic value.

So with a simple loop, we can calculate the CRC table. Easy peasy. Listing 2 has the simple details. After we have *that*, we can calculate the CRC value for a block of data, a byte at a time. Listing 3 shows this simple routine.

It's a bit messy to have one routine to calculate the table, and another to apply the table to

➤ *Listing 2: Calculating CRC table (XMODEM/CRC).*

```
procedure AACalc16BitCRCTable(var aTable : Taa16BitCRCTable;
  aMagicPoly : TaaCRC16);
const
  TopmostBitMask = $8000;
var
  i   : integer;
  Reg : TaaCRC16;
  bit : integer;
begin
  for i := 0 to 255 do begin
    Reg := i shl 8;
    for bit := 0 to 7 do begin
      if ((Reg and TopmostBitMask) <> 0) then
        Reg := (Reg shl 1) xor aMagicPoly
      else
        Reg := (Reg shl 1);
    end;
    aTable[i] := Reg;
  end;
end;
```

calculate the CRC for a block of data. We could have the table as a static compile-time constant, but that's a 512-byte table we're talking about. I think it would be better to design a CRC class. The Create constructor would be passed a magic polynomial as a parameter, and would allocate a table and generate it. The Destroy destructor would destroy it, of course. We'd also have a method for calculating the CRC of a block of data. We could have a routine that would write out the CRC table as a Delphi include file for example, so you could declare a static table.

There is something else to notice about the preceding argument which leads to a small optimization: instead of feeding in bits as we go along (they play no part in the determination of the XOR), all we need do is use the bits of the block to determine whether to XOR the magic polynomial or not, *as if they were fed into the top of the register directly*. The algorithm becomes:

1. Get the next bit from the block. If there is no next bit (ie we've exhausted the block), stop. The register holds the remainder.
2. XOR the top bit of the register with the next bit from the block.
3. If the result is 0, shift the register left by one. Go to step 1.
4. Otherwise, shift the register left by one bit, XOR the magic polynomial with the register. Go to step 1.

```
function AAGet16BitCRCTbl(var aBuffer; aBufLen : integer;
  const aTable : Taa16BitCRCTable) : TaaCRC16;
var
  i   : integer;
  Buf : TByteArray absolute aBuffer;
  Reg : TaaCRC16;
begin
  Reg := 0;  {initialise the register}
  {calculate the CRC}
  for i := 0 to pred(aBufLen) do
    Reg := aTable[byte(Reg shr 8) xor Buf[i]] xor (Reg shl 8);
  Result := Reg;  {return the register}
end;
```

➤ *Listing 3: Calculating CRC using table (XMODEM/CRC).*

Crazy

This new algorithm has a small gotcha though. If you think back to the original algorithm, it stopped when there were no more bits in the block. The last 16 bits were still in the register. With this new algorithm the last 16 bits were never in the register, it's as if we were using the original algorithm and after the block was exhausted we then fed in 16 zero bits to force the bits in the block all the way through. The extra zero bits wouldn't affect the XORing (any bit XOR zero equals the bit we had to begin with), but it would mean that the final CRC would have been calculated using all the bits in the block. This modified algorithm (feeding bits into the top of the register) can be extended to bytes at a time quite easily.

But we're not finished with the bit twiddling tricks yet. One of the original implementations of CRC was in hardware for checking receipt of a block of data that came over a phone line. The serial port actually sends bytes least significant bit first, so the hardware for calculating the CRC would get the bytes in bit-reversed order. It would calculate the CRC using these reversed bytes. When it came time to implement this in software, instead of reversing each byte and then using the usual algorithm, the original programmer decided to reverse everything else and leave the bytes alone. We've done the same ever since. So: the register is reversed; the magic polynomial is reversed; we shift the register right, instead of left; we feed in the bits in the bytes from the message into the lower bit of the register and this decides whether to XOR the reversed polynomial or not. Finally the CRC is returned in reversed form. We are no longer in Kansas, Toto.

Some CRC computations use this reversed scheme, some don't. The CRC-CCITT implementation uses reversed bits, whereas, as we've seen, the XMODEM/CRC version does not.

➤ *Listing 4: Full blown 32-bit CRC calculation.*

```
constructor TaaCRC32Calculator.Create(aMagicPoly : TaaCRC32;
  aReverseBits : boolean; aInitValue : TaaCRC32;
  aNotResult : boolean);
begin
  inherited Create;
  ccMagicPoly := aMagicPoly;
  ccReverseBits := aReverseBits;
  ccInitValue := aInitValue;
  ccNotResult := aNotResult;
end;
function TaaCRC32Calculator.GetCRCStd(var aBuffer;
  aBufLen : integer) : TaaCRC32;
var
  i   : integer;
  Buf : TByteArray absolute aBuffer;
  Reg : TaaCRC32;
  bit : integer;
  B   : byte;
  MagicPoly : TaaCRC32;
begin
  Reg := ccInitValue;  {initialise the register}
  {split the flow: first the case for feeding in bytes the
   least significant bit first}
  if ccReverseBits then begin
    {reverse the magic polynomial}
    MagicPoly := ReverseBits(ccMagicPoly, 32);
    {do for all bytes in the buffer...}
    for i := 0 to pred(aBufLen) do begin
      B := Buf[i];
      for bit := 0 to 7 do begin
        if ((Reg and 1) xor (B and 1)) <> 0 then
          Reg := (Reg shr 1) xor MagicPoly
        else
          Reg := (Reg shr 1);
        B := B shr 1;
      end;
    end;
  end
  {now the case for feeding in bytes the most significant
   bit first}
  else begin
    {do for all bytes in the buffer...}
    for i := 0 to pred(aBufLen) do begin
      B := Buf[i];
      for bit := 0 to 7 do begin
        if ((B shr 7) xor (Reg shr 31)) <> 0 then
          Reg := (Reg shl 1) xor ccMagicPoly
        else
          Reg := (Reg shl 1);
        B := B shl 1;
      end;
    end;
  end;
  {if required, not the result}
  if ccNotResult then
    Reg := not Reg;
  Result := Reg;  {return the register}
end;
```

```
procedure TaaCRC32Calculator.ccCreateTable;
const
  TopmostBitMask = $80000000;
var
  i   : integer;
  Reg : TaaCRC32;
  bit : integer;
begin
  New(ccTable);
  for i := 0 to 255 do begin
    if ccReverseBits then
      Reg := ReverseBits(i, 32)
    else
      Reg := TaaCRC32(i) shl 24;
    for bit := 0 to 7 do begin
      if ((Reg and TopmostBitMask) <> 0) then
        Reg := (Reg shl 1) xor ccMagicPoly
      else
        Reg := (Reg shl 1);
    end;
    if ccReverseBits then
      ccTable^[i] := ReverseBits(Reg, 32)
    else
      ccTable^[i] := Reg;
  end;
end;
```

```
function TaaCRC32Calculator.GetCRC(var aBuffer;
  aBufLen : integer) : TaaCRC32;
var
  i   : integer;
  Reg : TaaCRC32;
  Buf : TByteArray absolute aBuffer;
begin
  {if the CRC table hasn't yet been calculated, allocate it
   and do so}
  if (ccTable = nil) then
    ccCreateTable;
  {initialise the register}
  Reg := ccInitValue;
  {calculate the CRC}
  if ccReverseBits then
    for i := 0 to pred(aBufLen) do
      Reg := ccTable^[byte(Reg) xor Buf[i]] xor (Reg shr 8)
  else
    for i := 0 to pred(aBufLen) do
      Reg := ccTable^[byte(Reg shr 24) xor Buf[i]] xor (Reg
        shl 8);
  {if required, not the result}
  if ccNotResult then
    Reg := not Reg;
  {return the register}
  Result := Reg;
end;
```

➤ *Listing 5: Generating the table and computing a 32-bit CRC with it.*

The next bit of weirdness is the initial value for the register before we start dividing. We've been using 0 all along, because it follows directly from the mathematics. The other value used is -1 (or $FFFF). The reason this alternative (all 1 bits) is sometimes used is that, if the block starts with a bunch of zero bits, they will play no part in the CRC calculation. The initial values of the register will be all zero until we get the first 1 bit from the message. So by forcing the register to all 1s to begin with means that the CRC XOR calculations start straight away. In fact, it's a little less obvious than that if we were implementing the original division. The -1 is not exactly the original value of the register, it is XORed into the first four bytes of the message. With our snazzy new algorithm where we feed bits directly into the top bit of the register, we can set the initial value to -1 and it will have the same effect as XORing with the first 4 bytes of the message.

The final funky parameter for CRC calculations is that some implementations insist on the routine returning the one's complement of the final CRC value. To get the one's complement you either XOR the final value with $FFFF, or you NOT the final value (both are equivalent). In general, if the initial value of the register is -1, you NOT the final CRC.

The CRC-CCITT algorithm requires both the initial value of the register to be -1 and to NOT the final value.

Just Like You Said

We're still not quite done of course, we have to think about 32-bit CRCs. Luckily all the previous exposition applies just as well to the 32-bit world. We don't have to change anything, apart from making the register and CRC value a longint. Actually, to make compilation with Delphi 4 easier we'll define a `TaaCRC32` type which will be a longint, except for Delphi 4 (and later) where it will be a longword.

The main CRC used in the ZIP file format is known as CRC-32. The initial value of the register is -1, we need to NOT the final value, and we have to use the reversed algorithm. The magic polynomial is $04C11DB7. Another standard 32-bit CRC I found is the `AAL5` method. Here, the initial value of the register is -1, we need to NOT the final value, and we have to use the standard (non-reversed) algorithm. Again, the magic polynomial is $04C11DB7. The `AAL5` implementation requires the final CRC produced to be byte reversed (but not bit reversed). Listing 4 shows the 32-bit CRC calculation extracted from the CRC class on this month's disk; this method will correctly cater for all the different cases I mentioned a moment ago.

For a table-driven version, we can do the same process as before. The table is now 1Kb in size, being 256 longint values. Again we can create a static table, or we can generate it at runtime. Listing 5 has the table calculation and the CRC computation using it.

Let me say in closing that I found an inordinate number of different CRC implementations out there. They all tended to use the standard polynomials ($1021, $8005, and $04C11DB7), but some of them reversed the bits, some didn't, some used different initial values than the standard. Be warned.

On the accompanying diskette you'll find a class implementation of both 16- and 32-bit CRCs. It has methods for calculating the CRC of a complete block, and also for updating a CRC a byte at a time (of course, with the latter routine you are responsible for setting the

➤ *Table 1: Common 16-bit CRC implementations.*

Name:	CRC-CCITT	XMODEM/CRC	CRC-16
MagicPoly:	$1021	$1021	$8005
InitValue:	$FFFF	0	0
ReverseBits:	false	false	true
NotResult:	true	false	false

Name:	CRC-32	AAL5
MagicPoly:	$04C11DB7	$04C11DB7
InitValue:	$FFFFFFFF	$FFFFFFFF
ReverseBits:	true	false
NotResult:	true	true

➤ *Table 2: Common 32-bit CRC implementations.*

initial value of the CRC, and of NOTting the final CRC, if required). For fun, I added a method to each class to dump the CRC table as a Delphi include file, so you can write a small app to generate a static CRC table for another, larger, application. Tables 1 and 2 show the definitions of the standard CRC implementations.

I hope this article has been instructive. I must admit to having been ignorant of how all this stuff worked, especially given the terse nature of a typical table-driven calculation, and so it was a voyage of discovery for me. It took a good week until I was sure I understood how to get from the basic division to the bit-reversed, funky initial value, table-driven implementation. I hope I've managed to crystallize things for you too.

Julian Bucknall is checksummed, sealed and delivered. He's looking forward to that Future Love Paradise when his algorithms book is done. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

References

C Programmer's Guide to NetBIOS, W. David Schwaderer (Howard W Sams & Company)

The CRC Pitstop at www.ross.net/crc