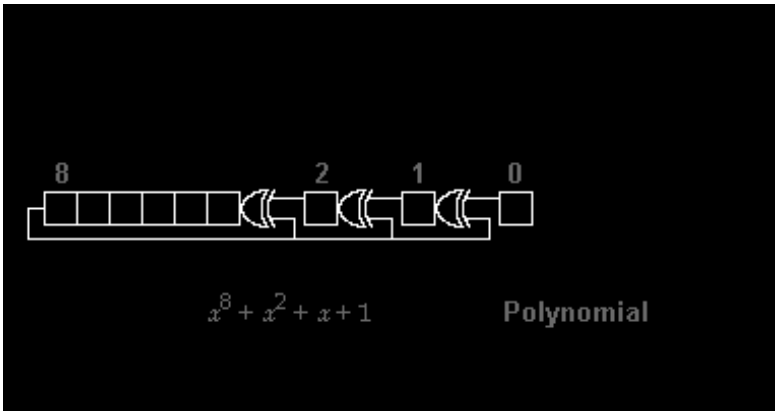
Computation of CRC

Computation of a cyclic redundancy check is derived from the mathematics of polynomial division, modulo two. In practice, it resembles long division of the binary message string, with a fixed number of zeroes appended, by the "generator polynomial" string except that exclusive OR operations replace subtractions. Division of this type is efficiently realised in hardware by a modified shift register,[1] and in software by a series of equivalent algorithms, starting with simple code close to the mathematics and becoming faster (and arguably more obfuscated[2]) through byte-wise parallelism and space-time tradeoffs.

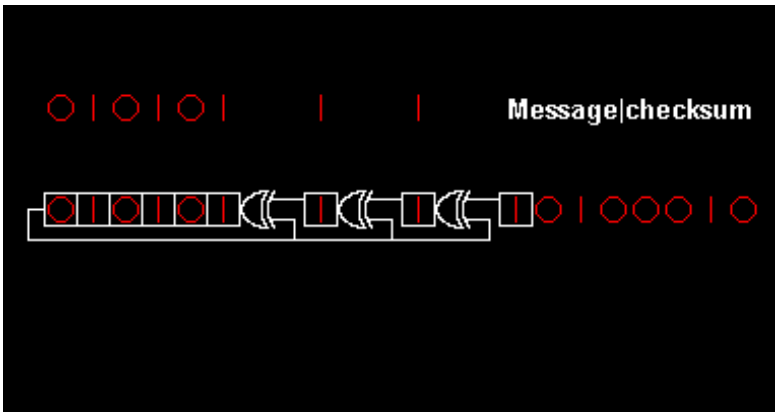
Various CRC standards extend the polynomial division algorithm by specifying an initial shift register value, a final exclusive OR step and, most critically, a bit ordering (endianness). As a result, the code seen in practice deviates confusingly from "pure" division,[2] and the register may shift left or right.



Example of generating an 8-bit CRC. The generator is a Galois type shift register with xor gates placed according to powers (white numbers) of *x* in the generator polynomial. The message stream may be any length. After it has been shifted through the register, followed by 8 zeroes, the result in the register is the checksum.

Example

As an example of implementing polynomial division in hardware, suppose that we are trying to compute an 8-bit CRC of an 8-bit message made of the ASCII character "W", which is binary 01010111_2, decimal 87_{10}, or hexadecimal 57_{16}. For illustration, we will use the CRC-8-ATM (HEC) polynomial $x^8 + x^2 + x + 1$. Writing the first bit transmitted (the coefficient of the highest power of x) on the left, this corresponds to the 9-bit string "100000111".

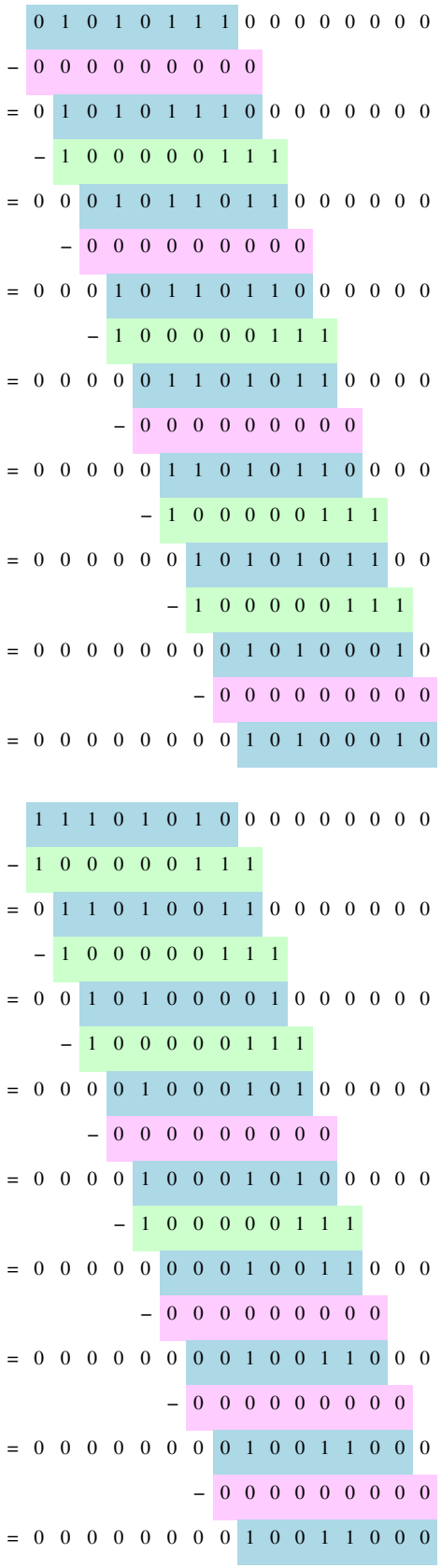


Checking received data with checksum. The received message is shifted through the same register as used in the generator, but the received checksum is attached to it instead of zeroes. Correct data yields the all-zeroes result; a corrupted bit in either the message or checksum would give a different result, warning that an error has occurred.

The byte value 57_{16} can be transmitted in two different orders, depending on the bit ordering convention used. Each one generates a different message polynomial $M(x)$. Msbit-first, this is $x^6 + x^4 + x^2 + x + 1$= 01010111, while lsbit-first, it is $x^7 + x^6 + x^5 + x^3 + x$= 11101010. These can then be multiplied by $x^8$to produce two 16-bit message polynomials $x^8 M(x)$.

Computing the remainder then consists of subtracting multiples of the generator polynomial $G(x)$. This is just like decimal long division, but even simpler because the only possible multiples at each step are 0 and 1, and the subtractions borrow "from infinity" instead of reducing the upper digits. Because we do not care about the quotient, there is no need to record it.

Most-significant bit first Least-significant bit first



Observe that after each subtraction, the bits are divided into three groups: at the beginning, a group which is all zero; at the end, a group which is unchanged from the original; and a shaded group in the middle which is "interesting". The "interesting" group is 8 bits long, matching the degree of the polynomial. Every step, the appropriate multiple of the polynomial is subtracted to make the zero group becomes one bit longer, and the unchanged group becomes one

bit shorter, until only the final remainder is left.

In the msbit-first example, the remainder polynomial is $x^7 + x^5 + x$. Converting to a hexadecimal number using the convention that the highest power of x is the msbit; this is $A2_{16}$. In the lsbit-first, the remainder is $x^7 + x^4 + x^3$. Converting to hexadecimal using the convention that the highest power of x is the lsbit, this is 19_{16}.

Implementation

Writing out the full message at each step, as done in the example above, is very tedious. Efficient implementations use an n-bit shift register to hold only the interesting bits. Multiplying the polynomial by x is equivalent to shifting the register by one place, as the coefficients do not change in value but only move up to the next term of the polynomial.

Here is a first draft of some pseudocode for computing an *n*-bit CRC. It uses a contrived composite data type for polynomials, where x is not an integer variable, but a constructor generating a *Polynomial* object that can be added, multiplied and exponentiated. To **xor** two polynomials is to add them, modulo two; that is, to exclusive OR the coefficients of each matching term from both polynomials.

```
function crc(bit array bitString[1..len], int len) {
     remainderPolynomial := polynomialForm(bitString[1..n])   // First n bits of the message
     // A popular variant complements remainderPolynomial here
     for i from 1 to len {
         remainderPolynomial := remainderPolynomial * x + bitString[i+n] * x^0   // Define bitString[k]=0 for k>len
         if coefficient of x^n of remainderPolynomial = 1 {
             remainderPolynomial := remainderPolynomial xor generatorPolynomial
         }
     }
     // A popular variant complements remainderPolynomial here
     return remainderPolynomial
 }
```

Code fragment 1: Simple polynomial division

Note that this example code avoids the need to specify a bit-ordering convention by not using bytes; the input `bitString` is already in the form of a bit array, and the `remainderPolynomial` is manipulated in terms of polynomial operations; the multiplication by x could be a left or right shift, and the addition of `bitString[i+n]` is done to the x^0coefficient, which could be the right or left end of the register.

This code has two disadvantages. First, it actually requires an *n*+1-bit register to hold the `remainderPolynomial` so that the x^n coefficient can be tested. More significantly, it requires the `bitString` to be padded with *n* zero bits.

The first problem can be solved by testing the x^{n-1}coefficient of the `remainderPolynomial` before it is multiplied by x.

The second problem could be solved by doing the last *n* iterations differently, but there is a more subtle optimization which is used universally, in both hardware and software implementations.

Because the XOR operation used to subtract the generator polynomial from the message is commutative and associative, it does not matter in what order the various inputs are combined into the `remainderPolynomial`. And specifically, a given bit of the `bitString` does not need to be added to the `remainderPolynomial` until the very last instant when it is tested to determine whether to `xor` with the `generatorPolynomial`.

This eliminates the need to preload the `remainderPolynomial` with the first *n* bits of the message, as well:

```
function crc(bit array bitString[1..len], int len) {
     remainderPolynomial := 0
     // A popular variant complements remainderPolynomial here
     for i from 1 to len {
         remainderPolynomial := remainderPolynomial xor (bitstring[i] * x^(n-1))
         if (coefficient of x^(n-1) of remainderPolynomial) = 1 {
             remainderPolynomial := (remainderPolynomial * x) xor generatorPolynomial
         } else {
             remainderPolynomial := (remainderPolynomial * x)
         }
     }
     // A popular variant complements remainderPolynomial here
     return remainderPolynomial
 }
```

Code fragment 2: Polynomial division with deferred message XORing

This is the standard bit-at-a-time hardware CRC implementation, and is well worthy of study; once you understand why this computes exactly the same result as the first version, the remaining optimizations are quite straightforward. If `remainderPolynomial` is only n bits long, then the x^n coefficients of it and of `generatorPolynomial` are simply discarded. This is the reason that you will usually see CRC polynomials written in binary with the leading coefficient omitted.

In software, it is convenient to note that while one may delay the `xor` of each bit until the very last moment, it is also possible to do it earlier. It is usually convenient to perform the `xor` a byte at a time, even in a bit-at-a-time implementation like this:

```
function crc(byte array string[1..len], int len) {
     remainderPolynomial := 0
     // A popular variant complements remainderPolynomial here
     for i from 1 to len {
         remainderPolynomial := remainderPolynomial xor polynomialForm(string[i]) * x^(n-8)
         for j from 1 to 8 {    // Assuming 8 bits per byte
             if coefficient of x^(n-1) of remainderPolynomial = 1 {
                 remainderPolynomial := (remainderPolynomial * x) xor generatorPolynomial
             } else {
                 remainderPolynomial := (remainderPolynomial * x)
             }
         }
     }
     // A popular variant complements remainderPolynomial here
     return remainderPolynomial
 }
```

Code fragment 3: Polynomial division with bytewise message XORing

This is usually the most compact software implementation, used in microcontrollers when space is at a premium over speed.

Bit ordering (Endianness)

When implemented in bit serial hardware, the generator polynomial uniquely describes the bit assignment; the first bit transmitted is always the coefficient of the highest power of x, and the last n bits transmitted are the CRC remainder $R(x)$, starting with the coefficient of x^{n-1} and ending with the coefficient of x^0, a.k.a. the coefficient of 1.

However, when bits are processed a byte at a time, such as when using parallel transmission, byte framing as in 8B/10B encoding or RS-232-style asynchronous serial communication, or when implementing a CRC in software, it is necessary to specify the bit ordering (endianness) of the data; which bit in each byte is considered "first" and will be the coefficient of the higher power of x.

If the data is destined for serial communication, it is best to use the bit ordering the data will ultimately be sent in. This is because a CRC's ability to detect burst errors is based on proximity in the message polynomial $M(x)$; if adjacent polynomial terms are not transmitted sequentially, a physical error burst of one length may be seen as a longer burst due to the rearrangement of bits.

For example, both IEEE 802 (ethernet) and RS-232 (serial port) standards specify least-significant bit first (little-endian) transmission, so a software CRC implementation to protect data sent across such a link should map the least significant bits in each byte to coefficients of the highest powers of x. On the other hand, floppy disks and most hard drives write the most significant bit of each byte first.

The lsbit-first CRC is slightly simpler to implement in software, so is somewhat more commonly seen, but many programmers find the msbit-first bit ordering easier to follow. Thus, for example, the XMODEM-CRC extension, an early use of CRCs in software, uses an msbit-first CRC.

So far, the pseudocode has avoided specifying the ordering of bits within bytes by describing shifts in the pseudocode as multiplications by x and writing explicit conversions from binary to polynomial form. In practice, the CRC is held in a standard binary register using a particular bit-ordering convention. In msbit-first form, the most significant binary bits will be sent first and so contain the higher-order polynomial coefficients, while in lsbit-first form, the least-significant binary bits contain the higher-order coefficients. The above pseudocode can be written in both forms. For concreteness, this uses the 16-bit CRC-16-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$:

```
// Most significant bit first (big-endian)
 // x^16+x^12+x^5+1 = (1) 0001 0000 0010 0001 = 0x1021
 function crc(byte array string[1..len], int len) {
     rem := 0
     // A popular variant complements rem here
     for i from 1 to len {
         rem := rem xor (string[i] leftShift (n-8))   // n = 16 in this example
         for j from 1 to 8 {   // Assuming 8 bits per byte
             if rem and 0x8000 {   // if leftmost (most significant) bit is set
                 rem := (rem leftShift 1) xor 0x1021
             } else {
                 rem := rem leftShift 1
             }
             rem := rem and 0xffff      // Trim remainder to 16 bits
         }
     }
     // A popular variant complements rem here
     return rem
 }
```

Code fragment 4: Shift register based division, MSB first

```
// Least significant bit first (little-endian)
 // x^16+x^12+x^5+1 = 1000 0100 0000 1000 (1) = 0x8408
 function crc(byte array string[1..len], int len) {
     rem := 0
     // A popular variant complements rem here
     for i from 1 to len {
         rem := rem xor string[i]
         for j from 1 to 8 {   // Assuming 8 bits per byte
             if rem and 0x0001 {   // if rightmost (least significant) bit is set
                 rem := (rem rightShift 1) xor 0x8408
             } else {
                 rem := rem rightShift 1
             }
         }
     }
     // A popular variant complements rem here
     return rem
 }
```

Code fragment 5: Shift register based division, LSB first

Note that the lsbit-first form avoids the need to shift `string[i]` before the `xor`. In either case, be sure to transmit the bytes of the CRC in the order that matches your chosen bit-ordering convention.

Parallel computation

Another common optimization uses a lookup table indexed by highest order coefficients of `rem` to perform the inner loop over 8 bits in fewer steps. A 256-entry lookup table is a particularly common choice, although using a 16-entry table twice per byte is very compact and still faster than the bit at a time version. This replaces the inner loop over `j` with

```
// Msbit-first
         rem = (rem leftShift 8) xor big_endian_table[(leftmost 8 bits of rem) rightShift (n-8)]


         // Lsbit-first
         rem = (rem rightShift 8) xor little_endian_table[rightmost 8 bits of rem]
```

Code fragment 6: Cores of table based division

One of the most commonly encountered CRC algorithms is known as **CRC-32**, used by (among others) Ethernet, FDDI, ZIP and other archive formats, and PNG image format. Its polynomial can be written msbit-first as 0x04C11DB7, or lsbit-first as 0xEDB88320. The W3C webpage on PNG includes an appendix with a short and simple table-driven implementation [3] in C of CRC-32. You will note that the code corresponds to the lsbit-first byte-at-a-time pseudocode presented here, and the table is generated using the bit-at-a-time code.

Parallel computation without table

Parallel update for a byte or a word at a time can also be done explicitly, without a table.[4] For each bit an equation is solved after 8 bits have been shifted in. The following tables list the equations for some commonly used polynomials, using following symbols:

c_i	CRC bit 7…0 (or 15…0) before update
r_i	CRC bit 7…0 (or 15…0) after update
d_i	input data bit 7…0
$e_i = d_i + c_i$	$e_p = e_7 + e_6 + \ldots + e_1 + e_0$ *(parity bit)*
$s_i = d_i + c_{i+8}$	$s_p = s_7 + s_6 + \ldots + s_1 + s_0$ *(parity bit)*

Bit-wise update equations for some CRC-8 polynomial after 8 bits have been shifted in

Polynomial:	$(x^7 + x^3 + 1) \times x$ *(left-shifted CRC-7-CCITT)*	$x^8 + x^5 + x^4 + 1$ *(CRC-8-Dallas/Maxim)*
Coefficients:	0x12 = (0x09 << 1) *(MSBF/normal)*	0x8c *(LSBF/reverse)*
	$r_0 \leftarrow 0$ $r_1 \leftarrow e_0 + e_4 + e_7$ $r_2 \leftarrow e_1 + e_5$ $r_3 \leftarrow e_2 + e_6$ $r_4 \leftarrow e_3 + e_7 + e_0 + e_4 + e_7$ $r_5 \leftarrow e_4 + e_1 + e_5$ $r_6 \leftarrow e_5 + e_2 + e_6$ $r_7 \leftarrow e_6 + e_3 + e_7$	$e_0 + e_4 + e_1 + e_0 + e_5 + e_2 + e_1$ $e_1 + e_5 + e_2 + e_1 + e_6 + e_3 + e_2 + e_0$ $e_2 + e_6 + e_3 + e_2 + e_0 + e_7 + e_4 + e_3 + e_1$ $e_3 + e_0 + e_7 + e_4 + e_3 + e_1$ $e_4 + e_1 + e_0$ $e_5 + e_2 + e_1$ $e_6 + e_3 + e_2 + e_0$ $e_7 + e_4 + e_3 + e_1$

C code fragments:	`uint8_t c, d, e, f, r;`
	`…`
	`e = c ^ d;`
	`f = e ^ (e >> 4) ^ (e >> 7);`
	`r = (f << 1) ^ (f << 4);`
		`uint8_t c, d, e, f, r;`
		`…`
		`e = c ^ d;`
		`f = e ^ (e << 3) ^ (e << 4) ^ (e << 6);`
		`r = f ^ (f >> 4) ^ (f >> 5);`

Bit-wise update equations for some CRC-16 polynomials after 8 bits have been shifted in

Polynomial:	$x^{16} + x^{12} + x^5 + 1$ *(CRC-16-CCITT)*		$x^{16} + x^{15} + x^2 + 1$ *(CRC-16-ANSI)*	
Coefficients:	0x1021 *(MSBF/normal)*	0x8408 *(LSBF/reverse)*	0x8005 *(MSBF/normal)*	0xa001 *(LSBF/reverse)*
	$r_0 \leftarrow s_4 + s_0$ $r_1 \leftarrow s_5 + s_1$ $r_2 \leftarrow s_6 + s_2$ $r_3 \leftarrow s_7 + s_3$ $r_4 \leftarrow s_4$ $r_5 \leftarrow s_5 + s_4 + s_0$ $r_6 \leftarrow s_6 + s_5 + s_1$ $r_7 \leftarrow s_7 + s_6 + s_2$ $r_8 \leftarrow c_0 + s_7 + s_3$ $r_9 \leftarrow c_1 + s_4$ $r_{10} \leftarrow c_2 + s_5$ $r_{11} \leftarrow c_3 + s_6$ $r_{12} \leftarrow c_4 + s_7 + s_4 + s_0$ $r_{13} \leftarrow c_5 + s_5 + s_1$ $r_{14} \leftarrow c_6 + s_6 + s_2$ $r_{15} \leftarrow c_7 + s_7 + s_3$	$c_8 + e_4 + e_0$ $c_9 + e_5 + e_1$ $c_{10} + e_6 + e_2$ $c_{11} + e_0 + e_7 + e_3$ $c_{12} + e_1$ $c_{13} + e_2$ $c_{14} + e_3$ $c_{15} + e_4 + e_0$ $e_0 + e_5 + e_1$ $e_1 + e_6 + e_2$ $e_2 + e_7 + e_3$ e_3 $e_4 + e_0$ $e_5 + e_1$ $e_6 + e_2$ $e_7 + e_3$	s_p $s_0 + s_p$ $s_1 + s_0$ $s_2 + s_1$ $s_3 + s_2$ $s_4 + s_3$ $s_5 + s_4$ $s_6 + s_5$ $c_0 + s_7 + s_6$ $c_1 + s_7$ c_2 c_3 c_4 c_5 c_6 $c_7 + s_p$	$c_8 + e_p$ c_9 c_{10} c_{11} c_{12} c_{13} $c_{14} + e_0$ $c_{15} + e_1 + e_0$ $e_2 + e_1$ $e_3 + e_2$ $e_4 + e_3$ $e_5 + e_4$ $e_6 + e_5$ $e_7 + e_6$ $e_p + e_7$ e_p

C code fragments:	`uint8_t  d, s, t;`
	`uint16_t c, r;`
	`…`
	`s = d ^ (c >> 8);`
	`t = s ^ (s >> 4);`
	`r = (c << 8) ^`
	`  t  ^`
	`(t << 5) ^`
	`(t << 12);`
		`uint8_t  d, e, f;`
		`uint16_t c, r;`
		`…`
		`e = c ^ d;`
		`f = e ^ (e << 4);`
		`r = (c >> 8) ^`
		`(f << 8) ^`
		`(f << 3) ^`
		`(f >> 4);`
			(s << 1);
			`r = (c << 8) ^`
			`(t << 15) ^`
			`t ^`
			`(t << 1);`
				(p << 8);
				`r = (c >> 8) ^`
				`(f << 6) ^`
				`(f << 7) ^`
				`(f >> 8);`

Two-step computation

As the CRC-32 polynomial has a large number of terms, when computing the remainder a byte at a time each bit depends on several bits of the previous iteration. In byte-parallel hardware implementations this calls for either multiple-input or cascaded XOR gates which increases propagation delay.

To maximise computation speed, an *intermediate remainder* can be calculated by passing the message through a 123-bit shift register. The polynomial is a carefully selected multiple of the standard polynomial such that the terms (feedback taps) are widely spaced, and no bit of the remainder is XORed more than once per byte iteration. Thus only two-input XOR gates, the fastest possible, are needed. Finally the intermediate remainder is divided by the standard polynomial in a second shift register to yield the CRC-32 remainder.[5]

One-pass checking

When appending a CRC to a message, it is possible to detach the transmitted CRC, recompute it, and verify the recomputed value against the transmitted one. However, a simpler technique is commonly used in hardware.

When the CRC is transmitted with the correct bit order (most significant terms first), a receiver can compute an overall CRC, over the message *and* the CRC, and if the CRC is correct, the result will be zero. This possibility is the reason that most network protocols that include a CRC do so *before* the ending delimiter; it is not necessary to know whether the end of the packet is imminent to check the CRC.

CRC variants

In practice, most standards specify presetting the register to all-ones and inverting the CRC before transmission. This has no effect on the ability of a CRC to detect changed bits, but gives it the ability to notice bits that are added to the message.

Preset to −1

The basic mathematics of a CRC accepts (considers as correctly transmitted) messages which, when interpreted as a polynomial, are a multiple of the CRC polynomial. If some leading 0 bits are prepended to such a message, they will not change its interpretation as a polynomial. This is equivalent to the fact that 0001 and 1 are the same number.

But if the message being transmitted does care about leading 0 bits, the inability of the basic CRC algorithm to detect such a change is undesirable. If it is possible that a transmission error could add such bits, a simple solution is to start with the `rem` shift register set to some non-zero value; for convenience, the all-ones value is typically used. This is mathematically equivalent to complementing (binary NOT) the first *n* bits of the message, where *n* are the number of bits in the CRC.

This does not affect CRC generation and checking in any way, as long as both generator and checker use the same initial value. Any non-zero initial value will do, and a few standards specify unusual values,[6] but the all-ones value (−1 in twos complement binary) is by far the most common.

Post-invert

The same sort of error can occur at the end of a message. Appending 0 bits to a message is equivalent to multiplying its polynomial by *x*, and if it was previously a multiple of the CRC polynomial, the result of that multiplication will be, as well. This is equivalent to the fact that, since 726 is a multiple of 11, so is 7260.

A similar solution can be applied at the end of the message, inverting the CRC register before it is appended to the message. Again, any non-zero change will do; inverting all the bits is simply the most common.

This has an effect on one-pass CRC checking; instead of producing a result of zero when the message is correct, it produces a fixed non-zero result. (To be precise, the result is the non-inverted CRC of the inversion pattern.) This is still straightforward to verify.

References

[1] Dubrova, Elena; and Mansouri, Shohreh S. (2012). "A BDD-Based Approach to Constructing LFSRs for Parallel CRC Encoding" (http://www.computer.org/portal/web/csdl/doi/10.1109/ISMVL.2012.20). *Proceedings of IEEE International Symposium on Multiple-Valued Logic*: 128–133. .

[2] Williams, Ross N. (1996-09-24). "A Painless Guide to CRC Error Detection Algorithms V3.00" (http://www.repairfaq.org/filipg/LINK/F_crc_v3.html). . Retrieved 2008-02-07.

[3] http://www.w3.org/TR/PNG/#D-CRCAppendix

[4] Jon Buller (1996-03-15). "[news:<31498ED0.7C0A@nortel.com> Re: 8051 and CRC-CCITT]". [news:comp.arch.embedded comp.arch.embedded]. Web link (http://groups.google.com/group/comp.arch.embedded/msg/cea9ca5da82017df). Retrieved 2009-07-27.

[5] Glaise, René J. (1997-01-20). "A two-step computation of cyclic redundancy code CRC-32 for ATM networks" (http://www.research.ibm.com/journal/rd/416/glaise.html). *IBM Journal of Research and Development* (Armonk, NY: IBM) **41** (6): 705. doi:10.1147/rd.416.0705. . Retrieved 2009-02-13.

[6] E.g. low-frequency RFID *TMS37157 data sheet* (http://www.ti.com/lit/gpn/tms37157), Texas Instruments, November 2009, p. 39, , retrieved 2011-04-15, "The CRC Generator is initialized with the value 0x3791 as shown in Figure 50."

External links

- 64-Bit CRC - Bitwise XOR Long-Division To Bytewise Table-Lookup (http://www.pathcom.com/~vadco/crc.html): Comprehensive documentation of CRC-Math, Procedure, and C-Code.

Article Sources and Contributors

Computation of CRC *Source*: http://en.wikipedia.org/w/index.php?oldid=527564497 *Contributors*: Arunkumar nonascii, Bobrayner, Bomazi, Cuddlyable3, Eugene-elgato, Gaius Cornelius, K5002, Matthijs, Mikechristoff, Nageh, Neonumbers, Regregex, SamB, XP1, Yfig, 18 anonymous edits

Image Sources, Licenses and Contributors

Image:CRC8-gen.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:CRC8-gen.gif *License*: Public domain *Contributors*: Bobmath, Magog the Ogre

Image:CRC8-rx.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:CRC8-rx.gif *License*: Public Domain *Contributors*: Bobmath

License