



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export

Create a book
Download as PDF
Printable version

Languages

العربية
Български
Català
Čeština
Dansk
Deutsch
Ελληνικά
Español
Euskara
فارسی
Français
한국어
Bahasa Indonesia
Italiano
עברית
Latviešu
日本語
Nederlands
Polski
Português
Română
Русский
Simple English
Slovenčina
Suomi
Svenska
Türkçe
Українська
Tiếng Việt
中文

Edit links

[Create account](#) [Log in](#)

[Article](#) [Talk](#)

[Read](#) [Edit](#) [View history](#)

Search

Cyclic redundancy check

From Wikipedia, the free encyclopedia

A **cyclic redundancy check** (**CRC**) is an **error-detecting code** commonly used in digital **networks** and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short *check value* attached, based on the remainder of a **polynomial division** of their contents. On retrieval the calculation is repeated, and corrective action can be taken against presumed data corruption if the check values do not match.

CRCs are so called because the *check* (data verification) value is a *redundancy* (it expands the message without adding **information**) and the **algorithm** is based on *cyclic codes*. CRCs are popular because they are simple to implement in binary **hardware**, easy to analyze mathematically, and particularly good at detecting common errors caused by **noise** in transmission channels. Because the check value has a fixed length, the **function** that generates it is occasionally used as a **hash function**.

The CRC was invented by **W. Wesley Peterson** in 1961; the 32-bit CRC function of Ethernet and many other standards is the work of several researchers and was published in 1975.

Contents

- 1 Introduction
- 2 Application
- 3 Data integrity
- 4 Computation
- 5 Mathematics
 - 5.1 Designing polynomials
- 6 Specification
- 7 Standards and common use
- 8 Implementations
- 9 See also
- 10 References
- 11 External links

Introduction [edit]

CRCs are based on the theory of **cyclic error-correcting codes**. The use of **systematic** cyclic codes, which encode messages by adding a fixed-length check value, for the purpose of error detection in communication networks, was first proposed by **W. Wesley Peterson** in 1961.^[1] Cyclic codes are not only simple to implement but have the benefit of being particularly well suited for the detection of **burst errors**, contiguous sequences of erroneous data symbols in messages. This is important because burst errors are common transmission errors in many **communication channels**, including magnetic and optical storage devices. Typically an *n*-bit CRC applied to a data block of arbitrary length will detect any single error burst not longer than *n* bits and will detect a fraction 1 − 2^{−*n*} of all longer error bursts.

Specification of a CRC code requires definition of a so-called **generator polynomial**. This polynomial becomes the **divisor** in a **polynomial long division**, which takes the message as the **dividend** and in which the **quotient** is discarded and the **remainder** becomes the result. The important caveat is that the polynomial **coefficients** are calculated according to the arithmetic of a **finite field**, so the addition operation can always be performed bitwise-parallel (there is no carry between digits). The length of the remainder is always less than the length of the generator polynomial, which therefore determines how long the result can be.

In practice, all commonly used CRCs employ the **Galois field** of two elements, **GF(2)**. The two elements are usually called 0 and 1, comfortably matching computer architecture.

A CRC is called an *n*-bit CRC when its check value is *n* bits. For a given *n*, multiple CRCs are possible, each with a different polynomial. Such a polynomial has highest degree *n*, which means it has *n* + 1 terms. In other words, the polynomial has a length of *n* + 1; its encoding requires *n* + 1 bits. Note that most polynomial specifications either drop the MSB or LSB bit, since they are always 1. The CRC and associated polynomial typically have a name of the form CRC-*n*-XXX as in the **table** below.

The simplest error-detection system, the **parity bit**, is in fact a trivial 1-bit CRC: it uses the generator polynomial *x* + 1 (two terms), and has the name CRC-1.

Application [edit]

A CRC-enabled device calculates a short, fixed-length binary sequence, known as the *check value* or *CRC*, for each block of data to be sent or stored and appends it to the data, forming a *codeword*. When a codeword is received or read, the device either compares its check value with one freshly calculated from the data block, or equivalently, performs a CRC on the whole codeword and compares the resulting check value with an expected *residue* constant. If the check values do not match, then the block contains a data error. The device may take corrective action, such as rereading the block or requesting that it be sent again. Otherwise, the data is assumed to be error-free (though, with some small probability, it may contain undetected errors; this is the fundamental nature of error-checking).^[2]

Data integrity [edit]

CRCs are specifically designed to protect against common types of errors on communication channels, where they can provide quick and reasonable assurance of the **integrity** of messages delivered. However, they are not suitable for protecting against intentional alteration of data.

Firstly, as there is no authentication, an attacker can edit a message and recompute the CRC without the substitution being detected. When stored alongside the data, CRCs and cryptographic hash functions by themselves do not protect against *intentional* modification of data. Any application that requires protection against such attacks must use cryptographic authentication mechanisms, such as **message authentication codes** or **digital signatures** (which are commonly based on **cryptographic hash** functions).

Secondly, unlike cryptographic hash functions, CRC is an easily reversible function, which makes it unsuitable for use in digital signatures.^[3]

Thirdly, CRC is a **linear function** with a property that **crc**(*x* ⊕ *y* ⊕ *z*) = **crc**(*x*) ⊕ **crc**(*y*) ⊕ **crc**(*z*): as a result, even if the CRC is encrypted with a **stream cipher** that uses **XOR** as its combining operation (or **mode of block cipher** which effectively turns it into a stream cipher, such as OFB or CFB), both the message and the associated CRC can be manipulated without knowledge of the encryption key; this was one of the well-known design flaws of the **Wired Equivalent Privacy** (WEP) protocol.^[4]

Computation [edit]

*Main article: **Computation of cyclic redundancy checks***

To compute an *n*-bit binary CRC, line the bits representing the input in a row, and position the (*n* + 1)-bit pattern representing the CRC's divisor (called a "**polynomial**") underneath the left-hand end of the row.

In this example, we shall encode 14 bits of message with a 3-bit CRC, with a polynomial *x*³ + *x* + 1. The polynomial is written in binary as the coefficients; a 3rd-order polynomial has 4 coefficients (1*x*³ + 0*x*² + 1*x* + 1). In this case, the coefficients are 1, 0, 1 and 1. The result of the calculation is 3 bits long.

Start with the message to be encoded:

```
11010011101100
```

This is first padded with zeros corresponding to the bit length *n* of the CRC. Here is the first calculation for computing a 3-bit CRC:

```
11010011101100 000 <--- input right padded by 3 bits
1011             <--- divisor (4 bits) = x³ + x + 1
-----
01100011101100 000 <--- result
```

The algorithm acts on the bits directly above the divisor in each step. The result for that iteration is the bitwise XOR of the polynomial divisor with the bits above it. The bits not above the divisor are simply copied directly below for that step. The divisor is then shifted one bit to the right, and the process is repeated until the divisor reaches the right-hand end of the input row. Here is the entire calculation:

```
11010011101100 000 <--- input right padded by 3 bits
1011             <--- divisor
01100011101100 000 <--- result (note the first four bits are the XOR with the divisor beneath, the rest of the bits are unchanged)
 1011            <--- divisor ...
```

```
00111011101100 000
 1011
0001011101100 000
 1011
00000001101100 000 <---- note that the divisor moves over to align with the next 1 in the dividend (since quotient for that step was zero)
 1011 (in other words, it doesn't necessarily move one bit per iteration)
00000000110100 000
 1011
00000000110000 000
 1011
00000000001110 000
 1011
00000000000101 000
 101 1
-----
00000000000000 100 <---- remainder (3 bits). Division algorithm stops here as quotient is equal to zero.
```

Since the leftmost divisor bit zeroed every input bit it touched, when this process ends the only bits in the input row that can be nonzero are the *n* bits at the right-hand end of the row. These *n* bits are the remainder of the division step, and will also be the value of the CRC function (unless the chosen CRC specification calls for some postprocessing).

The validity of a received message can easily be verified by performing the above calculation again, this time with the check value added instead of zeroes. The remainder should equal zero if there are no detectable errors.

```
11010011101100 100 <--- input with check value
1011 <--- divisor
01100011101100 100 <--- result
1011 <--- divisor ...
00111011101100 100

.....

00000000001110 100
 1011
00000000000101 100
 101 1
-----
0 <--- remainder
```

Mathematics [\[edit\]](#)

Main article: *Mathematics of cyclic redundancy checks*

Mathematical analysis of this division-like process reveals how to select a divisor that guarantees good error-detection properties. In this analysis, the digits of the bit strings are taken as the coefficients of a polynomial in some variable *x*—coefficients that are elements of the finite field **GF(2)**, instead of more familiar numbers. The set of binary polynomials is a mathematical **ring**.

Designing polynomials [\[edit\]](#)

The selection of the generator polynomial is the most important part of implementing the CRC algorithm. The polynomial must be chosen to maximize the error-detecting capabilities while minimizing overall collision probabilities.

The most important attribute of the polynomial is its length (largest degree(exponent) +1 of any one term in the polynomial), because of its direct influence on the length of the computed check value.

The most commonly used polynomial lengths are:

- 9 bits (CRC-8)
- 17 bits (CRC-16)
- 33 bits (CRC-32)
- 65 bits (CRC-64)

A CRC is called an *n*-bit CRC when its check value is *n*-bits. For a given *n*, multiple CRCs are possible, each with a different polynomial. Such a polynomial has highest degree *n*, and hence *n* + 1 terms (the polynomial has a length of *n* + 1). The remainder has length *n*. The CRC has a name of the form CRC-*n*-XXX.

The design of the CRC polynomial depends on the maximum total length of the block to be protected (data + CRC bits), the desired error protection features, and the type of resources for implementing the CRC, as well as the desired performance. A common misconception is that the "best" CRC polynomials are derived from either **irreducible polynomials** or irreducible polynomials times the factor 1 + *x*, which adds to the code the ability to detect all errors affecting an odd number of bits.^[9] In reality, all the factors described above should enter into the selection of the polynomial and may lead to a reducible polynomial. However, choosing a reducible polynomial will result in a certain proportion of missed errors, due to the quotient ring having **zero divisors**.

The advantage of choosing a **primitive polynomial** as the generator for a CRC code is that the resulting code has maximal total block length in the sense that all 1-bit errors within that block length have different remainders (also called **syndromes**) and therefore, since the remainder is a linear function of the block, the code can detect all 2-bit errors within that block length. If *r* is the degree of the primitive generator polynomial, then the maximal total block length is 2^{*r*} − 1, and the associated code is able to detect any single-bit or double-bit errors.^[9] We can improve this situation. If we use the generator polynomial *g*(*x*) = *p*(*x*)(1 + *x*) where *p*(*x*) is a primitive polynomial of degree *r* − 1, then the maximal total block length is 2^{*r*−1} − 1, and the code is able to detect single, double, triple and any odd number of errors.

A polynomial *g*(*x*) that admits other factorizations may be chosen then so as to balance the maximal total blocklength with a desired error detection power. The **BCH codes** are a powerful class of such polynomials. They subsume the two examples above. Regardless of the reducibility properties of a generator polynomial of degree *r*, if it includes the "+1" term, the code will be able to detect error patterns that are confined to a window of *r* contiguous bits. These patterns are called "error bursts".

Specification [\[edit\]](#)

The concept of the CRC as an error-detecting code gets complicated when an implementer or standards committee uses it to design a practical system. Here are some of the complications:

- Sometimes an implementation **prefixes a fixed bit pattern** to the bitstream to be checked. This is useful when clocking errors might insert 0-bits in front of a message, an alteration that would otherwise leave the check value unchanged.
- Usually, but not always, an implementation **appends *n* 0-bits** (*n* being the size of the CRC) to the bitstream to be checked before the polynomial division occurs. Such appending is explicitly demonstrated in the **Computation of CRC** article. This has the convenience that the remainder of the original bitstream with the check value appended is exactly zero, so the CRC can be checked simply by performing the polynomial division on the received bitstream and comparing the remainder with zero. Due to the associative and commutative properties of the exclusive-or operation, practical table driven implementations can obtain a result numerically equivalent to zero-appending without explicitly appending any zeroes, by using an equivalent,^[9] faster algorithm that combines the message bitstream with the stream being shifted out of the CRC register.
- Sometimes an implementation **exclusive-ORs a fixed bit pattern** into the remainder of the polynomial division.
- **Bit order**: Some schemes view the low-order bit of each byte as "first", which then during polynomial division means "leftmost", which is contrary to our customary understanding of "low-order". This convention makes sense when **serial-port** transmissions are CRC-checked in hardware, because some widespread serial-port transmission conventions transmit bytes least-significant bit first.
- **Byte order**: With multi-byte CRCs, there can be confusion over whether the byte transmitted first (or stored in the lowest-addressed byte of memory) is the least-significant byte (LSB) or the most-significant byte (MSB). For example, some 16-bit CRC schemes swap the bytes of the check value.
- **Omission of the high-order bit** of the divisor polynomial: Since the high-order bit is always 1, and since an *n*-bit CRC must be defined by an (*n* + 1)-bit divisor which **overflows** an *n*-bit **register**, some writers assume that it is unnecessary to mention the divisor's high-order bit.
- **Omission of the low-order bit** of the divisor polynomial: Since the low-order bit is always 1, authors such as Philip Koopman represent polynomials with their high-order bit intact, but without the low-order bit (the *x*⁰ or 1 term). This convention encodes the polynomial complete with its degree in one integer.

These complications mean that there are three common ways to express a polynomial as an integer: the first two, which are mirror images in binary, are the constants found in code; the third is the number found in Koopman's papers. *In each case, one term is omitted*. So the polynomial *x*⁴ + *x* + 1 may be transcribed as:

- 0x3 = 0b0011, representing *x*⁴+0*x*³ + 0*x*² + 1*x*¹ + 1*x*⁰ (MSB-first code)
- 0xC = 0b1100, representing 1*x*⁰ + 1*x*¹ + 0*x*² + 0*x*³+*x*⁴ (LSB-first code)
- 0x9 = 0b1001, representing 1*x*⁴ + 0*x*³ + 0*x*² + 1*x*¹+*x*⁰ (Koopman notation)

In the table below they are shown as:

Examples of CRC Representations			
Name	Normal	Reversed	Reversed reciprocal
CRC-4	0x3	0xC	0x9

Standards and common use [\[edit\]](#)

Numerous varieties of cyclic redundancy checks have been incorporated into [technical standards](#). By no means does one algorithm, or one of each degree, suit every purpose; Koopman and Chakravarty recommend selecting a polynomial according to the application requirements and the expected distribution of message lengths.^[7] The number of distinct CRCs in use has confused developers, a situation which authors have sought to address.^[9] There are three polynomials reported for CRC-12,^[7] sixteen conflicting definitions of CRC-16, and six of CRC-32.^[8]

The polynomials commonly applied are not the most efficient ones possible. Between 1993 and 2004, Koopman, Castagnoli and others surveyed the space of polynomials up to 16 bits,^[7] and of 24 and 32 bits.^{[9][10]} finding examples that have much better performance (in terms of [Hamming distance](#) for a given message size) than the polynomials of earlier protocols, and publishing the best of these with the aim of improving the error detection capacity of future standards.^[10] In particular, [iSCSI](#) and [SCTP](#) have adopted one of the findings of this research, the CRC-32C (Castagnoli) polynomial.

The design of the 32-bit polynomial most commonly used by standards bodies, CRC-32-IEEE, was the result of a joint effort for the [Rome Laboratory](#) and the Air Force Electronic Systems Division by Joseph Hammond, James Brown and Shyan-Shiang Liu of the [Georgia Institute of Technology](#) and Kenneth Brayer of the [MITRE](#) Corporation. The earliest known appearances of the 32-bit polynomial were in their 1975 publications: Technical Report 2956 by Brayer for MITRE, published in January and released for public dissemination through [DTIC](#) in August,^[11] and Hammond, Brown and Liu's report for the Rome Laboratory, published in May.^[12] Both reports contained contributions from the other team. During December 1975, Brayer and Hammond presented their work in a paper at the IEEE National Telecommunications Conference: the IEEE CRC-32 polynomial is the generating polynomial of a [Hamming code](#) and was selected for its error detection performance.^[13] Even so, the Castagnoli CRC-32C polynomial used in iSCSI or SCTP matches its performance on messages from 58 bits to 131 kbits, and outperforms it in several size ranges including the two most common sizes of Internet packet.^[10] The [ITU-T G.hn](#) standard also uses CRC-32C to detect errors in the payload (although it uses CRC-16-CCITT for PHY headers).

The table below lists only the polynomials of the various algorithms in use. Variations of a particular protocol can impose pre-inversion, post-inversion and reversed bit ordering as described above. For example, the CRC32 used in Gzip and Bzip2 use the same polynomial, but Gzip employs reversed bit ordering, while Bzip2 does not.^[8]

CRCs in proprietary protocols might be [obfuscated](#) by using a non-trivial initial value and a final XOR, but these techniques do not add cryptographic strength to the algorithm and can be [reverse engineered](#) using straightforward methods.^[14]

See [Polynomial representations of cyclic redundancy checks](#) for the algebraic representations of the polynomials for the CRCs below.

Name	Uses	Polynomial representations		
		Normal	Reversed	Reversed reciprocal
CRC-1	most hardware; also known as <i>parity bit</i>	0x1	0x1	0x1
CRC-4-ITU	G.704 ^[8]	0x3	0xC	0x9
CRC-5-EPC	Gen 2 RFID ^[15]	0x09	0x12	0x14
CRC-5-ITU	G.704 ^[8]	0x15	0x15	0x1A
CRC-5-USB	USB token packets	0x05	0x14	0x12
CRC-6-CDMA2000-A	mobile networks ^[16]	0x27	0x39	0x33
CRC-6-CDMA2000-B	mobile networks ^[16]	0x07	0x38	0x23
CRC-6-ITU	G.704 ^[8]	0x03	0x30	0x21
CRC-7	telecom systems, G.707 ^[8] , G.832 ^[8] , MMC , SD	0x09	0x48	0x44
CRC-7-MVB	Train Communication Network , IEC 60870-5 ^[17]	0x65	0x53	0x72
CRC-8		0xD5	0xAB	0xEA ^[7]
CRC-8-CCITT	L432.1 ^[8] ; ATM HEC , ISDN HEC and cell delineation	0x07	0xE0	0x83
CRC-8-Dallas/Maxim	1-Wire bus	0x31	0x8C	0x98
CRC-8-SAE J1850	AES3	0x1D	0xB8	0x8E
CRC-8-WCDMA	mobile networks ^{[16][18]}	0x9B	0xD9	0xCD ^[7]
CRC-10	ATM ; L610 ^[8]	0x233	0x331	0x319
CRC-10-CDMA2000	mobile networks ^[16]	0x3D9	0x26F	0x3EC
CRC-11	FlexRay ^[19]	0x385	0x50E	0x5C2
CRC-12	telecom systems ^{[20][21]}	0x80F	0xF01	0xC07 ^[7]
CRC-12-CDMA2000	mobile networks ^[16]	0xF13	0xC8F	0xF89
CRC-13-BBC	Time signal, Radio teleswitch ^[22]	0x1CF5	0x15E7	0x1E7A
CRC-15-CAN		0x4599	0x4CD1	0x62CC
CRC-15-MPT1327	^[23]	0x6815	0x540B	0x740A
Chakravarty	optimal for payloads ≤64 bits ^[17]	0x2F15	0xA8F4	0x978A
CRC-16-ARINC	ACARS applications ^[24]	0xA02B	0xD405	0xD015
CRC-16-CCITT	X.25 , V.41 , HDLC FCS , XMODEM , Bluetooth , PACTOR , SD , DigRF , many others; known as <i>CRC-CCITT</i>	0x1021	0x8408	0x8810 ^[7]
CRC-16-CDMA2000	mobile networks ^[16]	0xC867	0xE613	0xE433
CRC-16-DECT	cordless telephones ^[25]	0x0589	0x91A0	0x82C4
CRC-16-T10-DIF	SCSI DIF	0x8BB7 ^[26]	0xEDD1	0xC5DB
CRC-16-DNP	DNP , IEC 870 , M-Bus	0x3D65	0xA6BC	0x9EB2
CRC-16-IBM	Bisync , Modbus , USB , ANSI X3.28 ^[8] , SIA DC-07 , many others; also known as <i>CRC-16</i> and <i>CRC-16-ANSI</i>	0x8005	0xA001	0xC002
Fletcher	Used in Adler-32 A & B Checksums	<i>Not a CRC; see Fletcher's checksum</i>		
CRC-17-CAN	CAN FD ^[27]	0x1685B	0x1B42D	0x1B42D
CRC-21-CAN	CAN FD ^[27]	0x102899	0x132281	0x18144C
CRC-24	FlexRay ^[19]	0x5D6DCB	0xD3B6BA	0xAEB6E5
CRC-24-Radix-64	OpenPGP , RTCM104v3	0x864CFB	0xDF3261	0xC3267D
CRC-30	CDMA	0x2030B9C7	0x38E74301	0x30185CE3
Adler-32	Zlib	<i>Not a CRC; see Adler-32</i>		
CRC-32	HDLC , ANSI X3.66 , ITU-T V.42 , Ethernet , Serial ATA , MPEG-2 , PKZIP , Gzip , Bzip2 , PNG , ^[28] many others	0x04C11DB7	0xEDB88320	0x82608EDB ^[10]
CRC-32C (Castagnoli)	iSCSI , SCTP , G.hn payload, SSE4.2 , Btrfs , ext4 , Ceph	0x1EDC6F41	0x82F63B78	0x8F6E37A0 ^[10]
CRC-32K (Koopman)		0x741B8CD7	0xEB31D82E	0xBA0DC66B ^[10]
CRC-32Q	aviation; ADXM ^[29]	0x814141AB	0xD5828281	0xC0A0A0D5
CRC-40-GSM	GSM control channel ^{[30][31]}	0x004820009	0x9000412000	0x8002410004
CRC-64-ECMA	ECMA-182 ^[8] , XZ Utils	0x42F0E1EBA9EA3693	0xC96C5795D7870F42	0xA17870F5D4F51B49
CRC-64-ISO	HDLC , Swiss-Pro/TrEMBL ; considered weak for hashing ^[32]	0x000000000000001B	0xD800000000000000	0x800000000000000D

Implementations [\[edit\]](#)

- See also [\[edit\]](#)

- ## References [\[edit\]](#)

- ## External links [[edit](#)]

- | | Standards of Ecma International | [hide] |
|---|--|---------------------|
| Application Interfaces | ANSI escape code · Common Language Infrastructure · Office Open XML · OpenXPS | |
| File Systems (Tape) | Advanced Intelligent Tape · DDS · DLT · Super DLT · Holographic Versatile Disc · Linear Tape-Open (Ultrium-1) · VXA | |
| File Systems (Disk) | CD-ROM · CD File System (CDFS) · FAT (FAT12 · FAT16 · FAT16B) · FD · UDF · Ultra Density Optical · Universal Media Disc | |
| Graphics | Universal 3D | |
| Programming Languages | C++/CLI · C# · Eiffel · JavaScript (E4X, ECMAScript) | |
| Radio Link Interfaces | NFC · UWB | |
| Other | TC20 Electromagnetic Compatibility [1] ↗ · TC12 Product safety [2] ↗ | |
| List of Ecma standards (1961 - Present) | | |

Categories: [Binary arithmetic](#) | [Cyclic redundancy checks](#) | [Finite fields](#)

 **WIKIMEDIA**
project