



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
[日本語](#)
[Edit links](#)

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

Article [Talk](#)

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

Fletcher's checksum

From Wikipedia, the free encyclopedia

The **Fletcher checksum** is an [algorithm](#) for computing a [position-dependent checksum](#) devised by John G. Fletcher (1934-2012) at [Lawrence Livermore Labs](#) in the late 1970s.^[1] The objective of the Fletcher checksum was to provide error-detection properties approaching those of a [cyclic redundancy check](#) but with the lower computational effort associated with summation techniques.

Contents [\[hide\]](#)

- The algorithm
 - [Review of simple checksums](#)
 - [Weaknesses of simple checksums](#)
 - [The Fletcher checksum](#)
- [Fletcher-16](#)
- [Fletcher-32](#)
- [Fletcher-64](#)
- [Comparison with the Adler checksum](#)
- [Example calculation of the Fletcher-16 checksum](#)
- [Weaknesses](#)
- Implementation
 - [Straightforward](#)
 - [Check bytes](#)
 - [Optimizations](#)
 - [Bit and byte ordering \(endianness / network order\)](#)
- [References](#)
- [Notes](#)
- [External links](#)

The algorithm [\[edit\]](#)

Review of simple checksums [\[edit\]](#)

As with simpler checksum algorithms, the Fletcher checksum involves dividing the [binary data](#) word to be protected from errors into short "blocks" of bits and computing the [modular](#) sum of those blocks. (Note that the terminology used in this domain can be confusing. The data to be protected, in its entirety, is referred to as a "word" and the pieces into which it is divided are referred to as "blocks". It is tempting to think of a block of data divided into words, which gets the terms the wrong way round.)

As an example, the data may be a message to be transmitted consisting of 136 characters, each stored as an 8-bit [byte](#), making a data word of 1088 bits in total. A convenient block size would be 8 bits, although this is not required. Similarly, a convenient modulus would be 255, although, again, others could be chosen. So, the simple checksum is computed by adding together all the 8-bit bytes of the message, dividing by 255 and keeping only the remainder. (In practice, the [modulo operation](#) is performed during the summation to control the size of the result.) The checksum value is transmitted with the message, increasing its length to 137 bytes or 1096 bits. The receiver of the message can re-compute the checksum and compare it to the value received to determine whether the message has been altered by the transmission process.

Weaknesses of simple checksums [\[edit\]](#)

The first weakness of the simple checksum is that it is insensitive to the order of the blocks (bytes) in the data word (message). If the order is changed, the checksum value will be the same and the change will not be detected. The second weakness is that the universe of checksum values is small, being equal to the chosen modulus. In our example, there are only 255 possible checksum values, so it is easy to see that even random data has about a 0.4% probability of having the same checksum as our message.

The Fletcher checksum [\[edit\]](#)

Fletcher addresses both of these weaknesses by computing a second value along with the simple checksum.

This is the modular sum of the values taken by the simple checksum as each block of the data word is added to it. The modulus used is the same. So, for each block of the data word, taken in sequence, the block's value is added to the first sum and the new value of the first sum is then added to the second sum. Both sums start with the value zero (or some other known value). At the end of the data word, the modulus operator is applied and the two values are combined to form the Fletcher checksum value.

Sensitivity to the order of blocks is introduced because once a block is added to the first sum, it is then repeatedly added to the second sum along with every block after it. If, for example, two adjacent blocks become exchanged, the one that was originally first will be added to the second sum one fewer times and the one that was originally second will be added to the second sum one more time. The final value of the first sum will be the same but the second sum will be different, detecting the change to the message.

The universe of possible checksum values is now the square of the value for the simple checksum. In our example, the two sums each with 255 possible values result in 65025 possible values for the combined checksum.

Fletcher-16 [\[edit\]](#)

When the data word is divided into 8 bit blocks, as in the example above, two 8-bit sums result and are combined into a 16-bit Fletcher checksum. Usually, the second sum will be multiplied by 256 and added to the simple checksum, effectively stacking the sums side-by-side in a 16-bit word with the simple checksum at the least significant end. This algorithm is then called the Fletcher-16 checksum. The use of the modulus 255 is also generally implied.

The choice of modulus must obviously be such that the results will fit in the block size. 256 is therefore the largest possible modulus for Fletcher-16. It is a poor choice, however, as bits that overflow past bit 7 of the sum are simply lost. A modulus that takes the overflow bits and mixes them into the lower bits provides better error detection. The modulus should, however, be large so as to obtain the largest universe of checksum values. The value 255 takes the second consideration over the first, but has been found to have excellent performance.

Fletcher-32 [\[edit\]](#)

When the data word is divided into 16 bit blocks, two 16-bit sums result and are combined into a 32-bit Fletcher checksum. Usually, the second sum will be multiplied by 2^{16} and added to the simple checksum, effectively stacking the sums side-by-side in a 32-bit word with the simple checksum at the least significant end. This algorithm is then called the Fletcher-32 checksum. The use of the modulus 65,535 is also generally implied. The rationale for this choice is the same as for Fletcher-16.

Fletcher-64 [\[edit\]](#)

When the data word is divided into 32 bit blocks, two 32-bit sums result and are combined into a 64-bit Fletcher checksum. Usually, the second sum will be multiplied by 2^{32} and added to the simple checksum, effectively stacking the sums side-by-side in a 64-bit word with the simple checksum at the least significant end. This algorithm is then called the Fletcher-64 checksum. The use of the modulus 4,294,967,295 is also generally implied. The rationale for this choice is the same as for Fletcher-16 and Fletcher-32.

Comparison with the Adler checksum [\[edit\]](#)

The [Adler-32](#) checksum is a specialization of the Fletcher-32 checksum devised by [Mark Adler](#). The modulus selected (for both sums) is the prime number 65,521 (65,535 is divisible by 3, 5, 17 and 257). The first sum also begins with the value 1. The selection of a prime modulus results in improved "mixing" (error patterns are detected with more uniform probability, improving the probability that the least detectable patterns will be detected, which tends to dominate overall performance). However, the reduction in size of the universe of possible checksum values acts against this and reduces performance slightly. One study showed that Fletcher-32 outperforms Adler-32 in both performance and in its ability to detect errors. As modulo-65,535 addition is considerably simpler and faster to implement than modulo-65,521 addition, the Fletcher-32 checksum is generally a faster algorithm. ^[2]

Example calculation of the Fletcher-16 checksum [\[edit\]](#)

As an example, a Fletcher-16 checksum shall be calculated and verified for a byte stream of 0x01 0x02.

- C0_initial = 0
- C1_initial = 0

Byte	$C0 = C0_{prev} + B$	$C1 = C1_{prev} + C0$	Description
0x01	0x01	0x01	First byte fed in
0x02	0x03	0x04	Second byte fed in

The checksum is therefore 0x0403. It could be transmitted with the byte stream and be verified as such on the receiving end. Another option is to compute in a second step a pair of check bytes, which can be appended to the byte stream so that the resulting stream has a global fletcher-16 checksum value of 0.

The values of the checkbytes are computed as follows:

- $CB0 = 255 - ((C0 + C1) \bmod 255)$
- $CB1 = 255 - ((C0 + CB0) \bmod 255)$

where C0 and C1 are the result of the last step in the fletcher-16 computation.

In our case the checksum bytes are $CB0=0xF8$ and $CB1=0x04$. The transmitted byte stream is 0x01 0x02 0xF8 0x04. The receiver runs the checksum on all four bytes and calculates a passing checksum of 0x00 0x00, as illustrated below:

Byte	$C0 = C0_{prev} + B$	$C1 = C1_{prev} + C0$	Description
0x01	0x01	0x01	First byte fed in
0x02	0x03	0x04	Second byte fed in
$CB0 = 0xF8$	$0x03 + 0xF8 = 0xFB$	$0x04 + 0xFB = 0x00$	Checksum calculation - byte 1
$CB1 = 0x04$	$0xFB + 0x04 = 0x00$	$0x00 + 0x00 = 0x00$	Checksum calculation - byte 2

Weaknesses [\[edit\]](#)

The Fletcher checksum cannot distinguish between blocks of all 0 bits and blocks of all 1 bits. For example, if a 16-bit block in the data word changes from 0x0000 to 0xFFFF, the Fletcher-32 checksum remains the same. This also means a sequence of all 00 bytes has the same checksum as a sequence (of the same size) of all FF bytes.

Implementation [\[edit\]](#)

These examples assume [two's complement arithmetic](#), as Fletcher's algorithm will be incorrect on one's complement machines.

Straightforward [\[edit\]](#)

The below is a treatment on how to calculate the checksum including the check bytes; i.e., the final result should equal 0, given properly-calculated check bytes. The code by itself, however, will not calculate the check bytes.

An inefficient but straightforward implementation of a [C language function](#) to compute the Fletcher-16 checksum of an [array](#) of 8-bit data elements follows:

```
1 uint16_t Fletcher16( uint8_t* data, int count )
2 {
3     uint16_t sum1 = 0;
4     uint16_t sum2 = 0;
5     int index;
6
7     for( index = 0; index < count; ++index )
8     {
9         sum1 = (sum1 + data[index]) % 255;
10        sum2 = (sum2 + sum1) % 255;
11    }
12
13    return (sum2 << 8) | sum1;
14 }
```

On lines 3 and 4, the sums are 16-bit [variables](#) so that the additions on lines 9 and 10 will not [overflow](#). The [modulo operation](#) is applied to the first sum on line 9 and to the second sum on line 10. Here, this is done after each addition, so that at the end of the [for loop](#) the sums are always reduced to 8-bits. At the end of the input data, the two sums are combined into the 16-bit Fletcher checksum value and returned by the function on line

13.

Each sum is computed modulo 255 and thus remains less than 0xFF at all times. This implementation will thus never produce the checksum results 0x00FF, 0xFF00 or 0xFFFF. It can produce the checksum result 0x0000, which may not be desirable in some circumstances (e.g. when this value has been reserved to mean "no checksum has been computed").

Check bytes [\[edit\]](#)

Example source code for calculating the check bytes, using the above function, is as follows. The check bytes may be appended to the end of the data stream, with the c0 coming before the c1.

```
uint16_t csum;
uint8_t c0, c1, f0, f1;

csum = Fletcher16( data, length);
f0 = csum & 0xff;
f1 = (csum >> 8) & 0xff;
c0 = 0xff - (( f0 + f1) % 0xff);
c1 = 0xff - (( f0 + c0 ) % 0xff);
```

Optimizations [\[edit\]](#)

An optimized implementation in the [C programming language](#) operates as follows:

```
uint32_t fletcher32( uint16_t const *data, size_t words )
{
    uint32_t sum1 = 0xffff, sum2 = 0xffff;

    while (words) {
        unsigned tlen = words > 359 ? 359 : words;
        words -= tlen;
        do {
            sum2 += sum1 += *data++;
        } while (--tlen);
        sum1 = (sum1 & 0xffff) + (sum1 >> 16);
        sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    }
    /* Second reduction step to reduce sums to 16 bits */
    sum1 = (sum1 & 0xffff) + (sum1 >> 16);
    sum2 = (sum2 & 0xffff) + (sum2 >> 16);
    return sum2 << 16 | sum1;
}
```

A few tricks, well-known to implementers of the [IP checksum](#), are used here for efficiency:

- This reduces to the range 1..65535 rather than 0..65534. Modulo 65535, the values 65535 = 0xffff and 0 are equivalent, but it is easier to detect overflow if the former convention is used. This also provides the guarantee that the resultant checksum will never be zero, so that value is available for a special flag, such as "checksum not yet computed".
- $65536 \equiv 1 \pmod{65535}$, so the [end-around carry](#) expression `(x & 0xffff) + (x >> 16)` reduces `x` modulo 65535. Only doing it once is not guaranteed to be complete, but it will be in the range 1..0xffffe. A second repetition guarantees a fully reduced sum in the range of 1..0xffff.
- This uses a 32-bit accumulator to perform a number of sums before doing any modular reduction. The magic value 359 is the largest number of sums that can be performed without numeric overflow, given the possible initial starting value of `sum1 = 0x1ffffe`. Any smaller value is also permissible; 256 may be convenient in many cases.
- The limit is 360 if starting from `sum1 = 0xffff`, but the example code only partially reduces `sum1` between inner loops.
- For 8 bit checksums (with 16 bit accumulators) the maximum number of sums that can be performed before doing the modular reduction is 20^{[\[note 1\]](#)}^{[\[citation needed\]](#)}.

An efficient 8 bit implementation in the [C programming language](#) is as follows:

```

uint16_t fletcher16( uint8_t const *data, size_t bytes )
{
    uint16_t sum1 = 0xff, sum2 = 0xff;

    while (bytes) {
        size_t tlen = bytes > 20 ? 20 : bytes;
        bytes -= tlen;
        do {
            sum2 += sum1 += *data++;
        } while (--tlen);
        sum1 = (sum1 & 0xff) + (sum1 >> 8);
        sum2 = (sum2 & 0xff) + (sum2 >> 8);
    }
    /* Second reduction step to reduce sums to 8 bits */
    sum1 = (sum1 & 0xff) + (sum1 >> 8);
    sum2 = (sum2 & 0xff) + (sum2 >> 8);
    return sum2 << 8 | sum1;
}


```

Bit and byte ordering (endianness / network order) [\[edit\]](#)

As with any calculation that divides a binary data word into short blocks and treats the blocks as numbers, any two systems expecting to get the same result should preserve the ordering of bits in the data word. In this respect, the Fletcher checksum is no different from other checksum and CRC algorithms and needs no special explanation.

An ordering problem that is easy to envision occurs when the data word is transferred byte-by-byte between a big-endian system and a little-endian system and the Fletcher-32 checksum is computed. If blocks are extracted from the data word in memory by a simple read of a 16-bit unsigned integer, then the values of the blocks will be different in the two systems, due to the reversal of the byte order of 16-bit data elements in memory, and the checksum result will be different as a consequence. The implementation examples, above, do not address ordering issues so as not to obscure the checksum algorithm. Because the Fletcher-16 checksum uses 8-bit blocks, it is not affected by byte endianness.


References [\[edit\]](#)

- ↑ Fletcher, J. G. (January 1982). "An Arithmetic Checksum for Serial Transmissions". *IEEE Transactions on Communications*. COM-30 (1): 247–252. doi:10.1109/tcom.1982.1095369 .
- ↑ Theresa C. Maxino, Philip J. Koopman (January 2009). "The Effectiveness of Checksums for Embedded Control Networks"  (PDF). IEEE Transactions on Dependable and Secure Computing.

Notes [\[edit\]](#)

- ↑ Magic Numbers 20 and 359: If each term in the sum were added once only, the addition could repeat 254 times (16-bit) or 65534 times (32-bit) before reduction is required. Because of the repeated additions into `sum2`, if every addition caused a carry, the maximum excess added into `sum2` is a **triangular number** plus the number of iterations; these limits define such a sum which is less than the 254 or 65534 underlying limit.

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

- [RFC 905](#) - *ISO Transport Protocol Specification* describes the Fletcher checksum algorithm summing to zero.
- [RFC 1146](#) - *TCP Alternate Checksum Options* describes the Fletcher checksum algorithm for use with TCP.
- [RFC 905](#) - information about generating (as well as verifying) such a checksum in Annex B.
- [Performance of Checksums and CRCs over Real Data](#)
- [Maxino & Koopman](#)  - compares Adler, Fletcher, and CRC checksums
- [John Kodis](#) - When it comes to high-speed data verification, Fletcher's checksum algorithm can do the job.

Categories: Checksum algorithms

of [Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

