Consistent Overhead Byte Stuffing

Consistent Overhead Byte Stuffing (**COBS**) is an algorithm for encoding data bytes that results in efficient, reliable, unambiguous packet framing regardless of packet content, thus making it easy for receiving applications to recover from malformed packets. It employs a particular byte value, typically zero, to serve as a *packet delimiter* (a special value that indicates the boundary between packets). When zero is used as a delimiter, the algorithm replaces each zero data byte with a non-zero value so that no zero data bytes will appear in the packet and thus be misinterpreted as packet boundaries.

Byte stuffing is a process that transforms a sequence of data bytes that may contain 'illegal' or 'reserved' values (such as packet delimiter) into a potentially longer sequence that contains no occurrences of those values. The extra length of the transformed sequence is typically referred to as the overhead of the algorithm. HDLC framing is a well-known example, used particularly in PPP (see RFC 1662 § 4.2 (https://datatracker.ietf.org/doc/html/rfc1662#section-4.2)). Although HDLC framing has an overhead of <1% in the *average* case, it suffers from a very poor *worst*-case overhead of 100%; for inputs that consist entirely of bytes that require escaping, HDLC byte stuffing will double the size of the input.

The COBS algorithm, on the other hand, tightly bounds the worst-case overhead. COBS requires a minimum of 1 byte overhead, and a maximum of $\lceil n/254 \rceil$ bytes for n data bytes (one byte in 254, rounded up). Consequently, the time to transmit the encoded byte sequence is highly predictable, which makes COBS useful for real-time applications in which jitter may be problematic. The algorithm is computationally inexpensive, and in addition to its desirable *worst*-case overhead, its *average* overhead is also low compared to other unambiguous framing algorithms like HDLC.[1][2] COBS does, however, require up to 254 bytes of *lookahead*. Before transmitting its first byte, it needs to know the position of the first zero byte (if any) in the following 254 bytes.

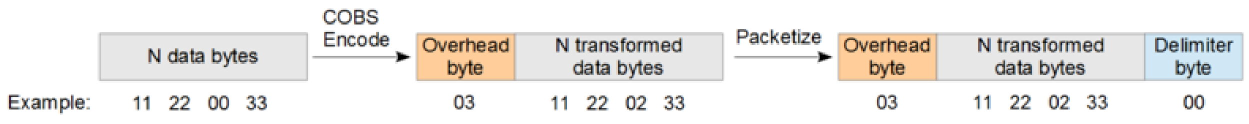
A 1999 Internet Draft proposed to standardize COBS as an alternative for HDLC framing in PPP, due to the aforementioned poor worst-case overhead of HDLC framing.[3]

Packet framing and stuffing

When packetized data is sent over any serial medium, some protocol is required to demarcate packet boundaries. This is done by using a framing marker, a special bit-sequence or character value that indicates where the boundaries between packets fall. Data stuffing is the process that transforms the packet data before transmission to eliminate all occurrences of the framing marker, so that when the receiver detects a marker, it can be certain that the marker indicates a boundary between packets.

COBS transforms an arbitrary string of bytes in the range [0,255] into bytes in the range [1,255]. Having eliminated all zero bytes from the data, a zero byte can now be used to unambiguously mark the end of the transformed data. This is done by appending a zero byte to the transformed data, thus forming a packet consisting of the COBS-encoded data (the *payload*) to unambiguously mark the end of the packet.

(Any other byte value may be reserved as the packet delimiter, but using zero simplifies the description.)



There are two equivalent ways to describe the COBS encoding process:

Prefixed block description

To encode some bytes, first append a zero byte, then break them into groups of either 254 non-zero bytes, or 0–253 non-zero bytes followed by a zero byte. Because of the appended zero byte, this is always possible.

Encode each group by deleting the trailing zero byte (if any) and prepending the number of non-zero bytes, plus one. Thus, each encoded group is the same size as the original, except that 254 non-zero bytes are encoded into 255 bytes by prepending a byte of 255.

As a special exception, if a packet ends with a group of 254 non-zero bytes, it is not necessary to add the trailing zero byte. This saves one byte in some situations.

Linked list description

First, insert a zero byte at the beginning of the packet, and after every run of 254 non-zero bytes. This encoding is obviously reversible. It is not necessary to insert a zero byte at the end of the packet if it happens to end with exactly 254 non-zero bytes.

Second, replace each zero byte with the offset to the next zero byte, or the end of the packet. Because of the extra zeros added in the first step, each offset is guaranteed to be at most 255.

Encoding examples

These examples show how various data sequences would be encoded by the COBS algorithm. In the examples, all bytes are expressed as hexadecimal values, and encoded data is shown with text formatting to illustrate various features:

- **Bold** indicates a data byte which has not been altered by encoding. All non-zero data bytes remain unaltered.
- Green indicates a zero data byte that was altered by encoding. All zero data bytes are replaced during encoding by the offset to the following zero byte (i.e. one plus the number of non-zero bytes that follow). It is effectively a pointer to the next packet byte that requires interpretation: if the addressed byte is non-zero then it is the following group header byte zero data byte that points to the next byte requiring interpretation; if the addressed byte is zero then it is the end of packet.
- Red is an overhead byte which is also a group header byte containing an offset to a following group, but does not correspond to a data byte. These appear in two places: at the beginning of every encoded packet, and after every group of 254 non-zero bytes.
- A blue zero byte appears at the end of every packet to indicate end-of-packet to the data receiver. This packet delimiter byte is not part of COBS proper; it is an additional framing byte that is appended to the encoded output.

Example	Unencoded data (hex)	Encoded with COBS (hex)
1	00	01 01 00
2	00 00	01 01 01 00
3	00 11 00	01 02 **11** 01 00
4	11 22 00 33	03 **11 22** 02 **33** 00
5	11 22 33 44	05 **11 22 33 44** 00
6	11 00 00 00	02 **11** 01 01 01 00
7	01 02 03 ... FD FE	FF **01 02 03 ... FD FE** 00
8	00 01 02 ... FC FD FE	01 FF **01 02 ... FC FD FE** 00
9	01 02 03 ... FD FE FF	FF **01 02 03 ... FD FE** 02 **FF** 00
10	02 03 04 ... FE FF 00	FF **02 03 04 ... FE FF** 01 01 00
11	03 04 05 ... FF 00 01	FE **03 04 05 ... FF** 02 **01** 00

Below is a diagram using example 4 from above table, to illustrate how each modified data byte is located, and how it is identified as a data byte or an end of frame byte.

```
   [OHB]                                : Overhead byte (Start of frame)
     3+ -------------->|                : Points to relative location of first zero symbol
                       2+-------->|     : Is a zero data byte, pointing to next zero
symbol
                                  [EOP] : Location of end-of-packet zero symbol.
     0     1     2     3     4    5     : Byte Position
     03    11    22    02    33   00    : COBS Data Frame
           11    22    00    33         : Extracted Data

OHB = Overhead Byte (Points to next zero symbol)
EOP = End Of Packet
```

Examples 7 through 10 show how the overhead varies depending on the data being encoded for packet lengths of 255 or more.

Implementation

The following code implements a COBS encoder and decoder in the C programming language:

```c
#include <stddef.h>
#include <stdint.h>
#include <assert.h>

/** COBS encode data to buffer
    @param data Pointer to input data to encode
    @param length Number of bytes to encode
    @param buffer Pointer to encoded output buffer
    @return Encoded buffer length in bytes
    @note Does not output delimiter byte
*/
size_t cobsEncode(const void *data, size_t length, uint8_t *buffer)
{
    assert(data && buffer);

    uint8_t *encode = buffer; // Encoded byte pointer
    uint8_t *codep = encode++; // Output code pointer
    uint8_t code = 1; // Code value

    for (const uint8_t *byte = (const uint8_t *)data; length--; ++byte)
    {
        if (*byte) // Byte not zero, write it
            *encode++ = *byte, ++code;

        if (!*byte || code == 0xff) // Input is zero or block completed, restart
        {
            *codep = code, code = 1, codep = encode;
            if (!*byte || length)
                ++encode;
        }
    }
    *codep = code; // Write final code value

    return (size_t)(encode - buffer);
}

/** COBS decode data from buffer
    @param buffer Pointer to encoded input bytes
    @param length Number of bytes to decode
    @param data Pointer to decoded output data
    @return Number of bytes successfully decoded
    @note Stops decoding if delimiter byte is found
*/
size_t cobsDecode(const uint8_t *buffer, size_t length, void *data)
{
    assert(buffer && data);

    const uint8_t *byte = buffer; // Encoded input byte pointer
    uint8_t *decode = (uint8_t *)data; // Decoded output byte pointer

    for (uint8_t code = 0xff, block = 0; byte < buffer + length; --block)
    {
        if (block) // Decode block byte
            *decode++ = *byte++;
        else
        {
            block = *byte++;             // Fetch the next block length
            if (block && (code != 0xff)) // Encoded zero, write it unless it's delimiter.
                *decode++ = 0;
            code = block;
            if (!code) // Delimiter code found
                break;
        }
    }

    return (size_t)(decode - (uint8_t *)data);
}
```

See also

- Bit stuffing
- Serial Line Internet Protocol

References

1. Cheshire, Stuart; Baker, Mary (April 1999). "Consistent Overhead Byte Stuffing" (http://www.stuartcheshire.org/papers/COBSforToN.pdf) (PDF). *IEEE/ACM Transactions on Networking*. **7** (2): 159–172. CiteSeerX 10.1.1.108.3143 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.108.3143) . doi:10.1109/90.769765 (https://doi.org/10.1109%2F90.769765) . S2CID 47267776 (https://api.semanticscholar.org/CorpusID:47267776) . Retrieved November 30, 2015.
2. Cheshire, Stuart; Baker, Mary (17 November 1997). *Consistent Overhead Byte Stuffing* (http://conferences.sigcomm.org/sigcomm/1997/papers/p062.pdf) (PDF). ACM SIGCOMM '97. Cannes. Retrieved November 23, 2010.
3. Carlson, James; Cheshire, Stuart; Baker, Mary (November 1997). *PPP Consistent Overhead Byte Stuffing (COBS)* (https://datatracker.ietf.org/doc/html/draft-ietf-pppext-cobs-00.txt) . I-D draft-ietf-pppext-cobs-00.txt.

External links

- Python implementation (https://pypi.python.org/pypi/cobs)
- Alternate C implementation (https://github.com/cmcqueen/cobs-c)
- Another implementation in C (https://web.archive.org/web/20180719005741/http://www.jacquesf.com/2011/03/consistent-overhead-byte-stuffing/)
- Consistent Overhead Byte Stuffing—Reduced (COBS/R) (https://pythonhosted.org/cobs/cobsr-intro.html)

- A patent describing a scheme with a similar result but using a different method (https://patents.google.com/patent/US9438411B1)