Consistent Overhead Byte Stuffing

Stuart Cheshire and Mary Baker
Computer Science Department, Stanford University
Stanford, California 94305, USA

{cheshire,mgbaker}@cs.stanford.edu

Abstract

Byte stuffing is a process that transforms a sequence of data bytes that may contain 'illegal' or 'reserved' values into a potentially longer sequence that contains no occurrences of those values. The extra length is referred to in this paper as the overhead of the algorithm.

To date, byte stuffing algorithms, such as those used by SLIP [RFC1055], PPP [RFC1662] and AX.25 [ARRL84], have been designed to incur low average overhead but have made little effort to minimize worst case overhead.

Some increasingly popular network devices, however, care more about the worst case. For example, the transmission time for ISM-band packet radio transmitters is strictly limited by FCC regulation. To adhere to this regulation, the practice is to set the maximum packet size artificially low so that no packet, even after worst case overhead, can exceed the transmission time limit.

This paper presents a new byte stuffing algorithm, called Consistent Overhead Byte Stuffing (COBS), that tightly bounds the worst case overhead. It guarantees in the worst case to add no more than one byte in 254 to any packet. Furthermore, the algorithm is computationally cheap, and its average overhead is very competitive with that of existing algorithms.

1. Introduction

The purpose of byte stuffing is to convert data packets into a form suitable for transmission over a serial medium like a telephone line. Boundaries between packets are marked by a special reserved byte value, and byte stuffing ensures, at the cost of a potential increase in packet size, that this reserved value does not inadvertently appear in the body of any transmitted packet. In general, some added overhead (in terms of additional bytes transmitted over the serial medium) is inevitable if we are to perform byte stuffing without loss of information.

Current byte stuffing algorithms, such as those used by SLIP [RFC1055], PPP [RFC1662] and AX.25 [ARRL84], have potentially high variance in per-packet overhead. For example, using PPP byte stuffing, the overall overhead (averaged over a large number of packets) is typically 1% or less, but individual packets can increase in size by as much as 100%.

While this variability in overhead may add some jitter and unpredictability to network behavior, it has only minor implications for running PPP over a telephone modem. The IP host using the PPP protocol has only to make its transmit and receive serial buffers twice as big as the largest IP packet it expects to send or receive. The modem itself is a connection-oriented device and is only concerned with transmitting an unstructured stream of byte values. It is unaware of the concept of packet boundaries, so the size of the packets it has to send does not have any direct design implications for the modem.

In contrast, new devices are now becoming available, particularly portable wireless devices, that are packet-oriented, not circuit-oriented. Unlike telephone modems these devices are not insensitive to the packet sizes they have to send. Channel-hopping packet radios that operate in the unlicensed ISM (Industrial/Scientific/Medical) bands under the FCC Part 15 rules [US94-15] have a maximum transmission time which they are not allowed to exceed. Unexpectedly doubling the size of a packet could result in a packet that is too big to transmit legally. Using PPP encoding, the only way to be certain that no packets will exceed the legal limit is to set the IP MTU (maximum transmission unit) to half the device's true MTU, despite the fact that it is exceedingly rare to encounter a packet that actually doubles in size when encoded. Drastically reducing the IP MTU in this way can significantly degrade the performance seen by the end-user.

Although, as seen from experiments presented in this paper, packets that actually double in size rarely occur naturally, it is not acceptable to ignore the possibility of their occurrence. Without a factor-of-two safety margin, the network device would be open to malicious attack through artificially constructed pathological packets. An attacker could use such packets to exploit the device's inability to send and/or receive worst-case packets, causing mischief ranging from simple denial of service attacks to the much more serious potential for exploiting finger-dæmon-style run-off-the-end-of-the-array bugs [RFC1135].

This problem could be solved in a variety of ways. One solution would be to set the IP MTU close to the underlying device MTU, and use link-layer fragmentation and reassembly for packets that exceed the limit when encoded. While this could work, requiring link-layer software to perform fragmentation and reassembly is a substantial burden we would rather not impose on driver writers. Fragmentation and reassembly code has proved notoriously difficult to get right, as evidenced by the recent spate of Internet "Ping of death" attacks exploiting a reassembly bug that exists in many implementations of IP [CA-96.26]. Link-layer fragmentation would also add protocol overhead, because the link-layer packet headers would have to contain additional fields to support the detection, ordering, and reassembly of fragments at the receiving end.

It might also be possible to use IP's own fragmentation and reassembly mechanisms, but this solution also has problems. One problem is that in current networking software implementations IP fragmentation occurs before the packet is handed to the device driver software for transmission. There is no mechanism for the device driver to hand the packet back to IP with a message saying, "Sorry, I failed, can you please refragment this packet and try again?" Also, some software goes to great lengths to perform path MTU discovery in order to send optimally sized packets. It would be difficult for such software to cope with an IP implementation where the MTU of a device may vary for each packet. Finally, it is unclear how this mechanism could handle IP packets that have the "Don't Fragment" header bit set.

Given these problems it would be much better if there were a byte stuffing algorithm that did not have such inconsistent behavior. This paper presents a new algorithm, called Consistent Overhead Byte Stuffing (COBS), which can be relied upon to encode all packets efficiently, regardless of their contents. It is simple to understand, computationally cheap, and easy to implement in software. All packets up to 254 bytes in length are encoded with an overhead of exactly one byte. For packets over 254 bytes in length the overhead is at most one byte for every 254 bytes of packet data. The maximum overhead can be calculated as 0.4% of the packet size, rounded up to a whole number of bytes.

Using this algorithm, the IP MTU can be set to within 0.4% of the underlying device's maximum packet size without fear of any packet inadvertently exceeding that limit. This gives better end-user performance, because now any given piece of hardware can consistently send larger IP packet payloads.

The rest of the paper proceeds as follows:

In the next section we give more detailed background information on the subject of packet framing and byte stuffing.

Section 3 describes COBS and Sections 4 and 5 compare its cost to using PPP to encode the same data packets. In the least favourable case for our algorithm, network traffic consisting predominantly of small packets, COBS is found to add less than 0.5% additional average overhead compared to PPP. Although this is a small price for the performance benefit of being able to use much larger packets, it is possible to eliminate even this cost. We present a trivial modification to COBS, called Zero Pair Elimination (ZPE), that improves on straight COBS's average performance at the expense of a fractionally higher worst case bound. In addition to having a very low bound on worst-case overhead, COBS/ZPE also achieves an average overhead lower than PPP's, even for small-packet traffic.

In Section 4 we consider the expected overhead from a theoretical point of view, for data consisting of uniformly distributed random eight-bit values. Of course real network traffic often does not have a uniform distribution of byte values, so in Section 5 we present experimental results comparing COBS with PPP for real network traffic. Section 6 presents our conclusions.

2. Background: Packet Framing and Data Stuffing

When packet data is sent over any serial medium, a protocol is needed by which to demark packet boundaries. This is done by using a special value or signal. That marker should be one that never occurs within the body of any packet, so that when the receiver detects that special marker, it knows, without any ambiguity, that it does indeed indicate a boundary between packets.

In hardware systems low-level framing protocols are common. For example, in IBM Token Ring the electrical signalling on the wire uses Differential

Manchester Encoding, and packet boundaries are marked by a violation of those encoding rules [IEEE802.5]. Making the packet delimiter an illegal signal value is a very simple way to ensure that no packet payload can inadvertently generate that signal on the cable. This approach is not a universal solution, because not all hardware supports deliberate generation of protocol violations, and even on hardware that does, use of this facility is usually restricted to the lowest levels of the driver firmware.

An alternative is to use a legal value as the frame delimiter. The drawback of this method is that the protocol needs to ensure that the frame delimiter value never occurs within the body of the packet. One approach is simply to prohibit the frame delimiter value from being used in the data of any packet, but this has the disadvantage that the communication protocol is no longer 'transparent' to the higher layers — the software using it has to be aware of which character values it can use and which it cannot.

A better approach is to allow the higher layers of software to send any character values they wish, and to make the framing protocol software responsible for transforming any data it is given into a form that contains no reserved character values and is suitable for transmission. Whatever transformation it performs must be reversible, so that the communications software at the receiving end can recover the original payload from the encoded form that is transmitted. The mechanisms generally used to do this are called bit stuffing algorithms (which operate on a bit-by-bit basis) or byte stuffing algorithms (which operate on a byte at a time).

Both bit stuffing and byte stuffing in general increase the size of the data being sent. The amount of increase depends on the patterns of values that appear in the original data and can vary from no overhead at all to doubling the packet size (in the worst case for PPP). These are discussed in more detail below.

HDLC [ECMA40] uses a bit stuffing scheme. It uses the binary sequence 01111110, called the Flag Sequence, to mark boundaries between packets. To eliminate this pattern from the data, the following transformation is used: whenever the transmitter observes five ones in a row, it inserts a zero immediately following. This eliminates the possibility of six ones ever occurring inadvertently in the data. The receiver performs the reverse process: After observing five ones in a row, if the next binary digit is a zero it is deleted, and if it is a one then the receiver recognizes it as one of the special framing patterns. This process of inserting extra zeroes ('bit stuffing' or 'zero insertion') increases the transmitted size of the data. In the worse case, for data that consists entirely of binary ones, HDLC framing can add 20% to the transmitted size of the data.

This kind of bit-level manipulation is easy to implement in the hardware of a serial transmitter, but is not easy to implement efficiently in software. Software gets efficiency from working in units of 8, 32, or more bits at a time, using on-chip registers and wide data buses. Algorithms that are specified in terms of individual bits can be hard to implement efficiently in software because it can be difficult to take good advantage of the processor's ability to operate on bytes or words at a time. For this reason it is more common for software algorithms to use byte stuffing.

PPP uses a byte stuffing scheme [RFC1662]. It uses a byte with value 0x7E (the same as the HDLC Flag Sequence) to mark boundaries between packets. To eliminate this value from the data payload, the following transformation is used: everywhere that 0x7E appears in the data it is replaced with the two-character sequence 0x7D,0x5E. 0x7D is called the Control Escape byte. Everywhere that 0x7D appears in the data it is replaced with the two-character sequence 0x7D,0x5D. The receiver performs the reverse process: whenever it sees the Control Escape value (0x7D) it discards that byte and XORs the following byte with 0x20 to recreate the original input.

On average this byte stuffing does reasonably well, increasing the size of purely random data by a little under 1%, but in the worst case it can double the size of the data being encoded. This is much worse than HDLC's worst case, but PPP byte stuffing has the advantage that it can be implemented reasonably efficiently in software.

PPP's byte stuffing mechanism, in which the offending byte is prefixed with 0x7D and XORed with 0x20, allows multiple byte values to be eliminated from a packet. For example, PPP byte stuffing can facilitate communication over a non-transparent network by eliminating all ASCII control characters (0x00-0x1F) from the transmitted packet data. It is equally possible to do this using a COBS-type algorithm, but the subject of this paper is the minimal byte stuffing necessary to facilitate reliable unambiguous packet framing, not extensive byte stuffing to compensate for non-transparency of the underlying network.

A more exhaustive treatment of framing and data stuffing can be found in [Che97].

3. Consistent Overhead Byte Stuffing Algorithms

We first provide an overview of the encoding used by COBS and then describe the procedure for performing that encoding. We conclude this section by describing

```
01
02 x
03 x y
0C H e l l o   W o r l d
```

Figure 1. Example Code Blocks

COBS takes its input data and encodes it as a series of variable length code blocks. Each code block begins with a single code byte (shown shaded), followed by zero or more data bytes.

some of the properties and implications of using COBS, particularly the fact that COBS can also eliminate the overhead of lower-level byte framing and bit stuffing.

3.1 Overview

COBS performs a reversible transformation on a data packet to eliminate a particular byte value from it. We pick zero as the value to eliminate, because zeroes are common in computer data and COBS performs better when it has many bytes to eliminate. However, with only trivial changes any desired value may be eliminated, as described below.

COBS takes its input data and encodes it as a series of variable length code blocks. Each code block begins with a single code byte, followed by zero or more data bytes. The number of data bytes is determined by the code byte. Figure 1 shows some examples of valid code blocks.

Apart from one exception, the meaning of each code block is that it represents the sequence of data bytes contained within the code block, *followed by an implicit zero*. The zero is implicit — it is not actually contained within the sequence of data bytes in the code block. That would defeat our purpose, which is that the output data should contain no zero values. The exception mentioned above is code 0xFF, which represents a run of 254 non-zero data bytes *without* an implicit zero on the end. This code acts as a kind of 'fail-safe' or 'escape hatch', allowing COBS to encode long sequences of bytes that do not contain any zeroes at all, which it would not otherwise be able to do. The meanings of the various code values are summarized in Table 1.

COBS has the property that the byte value zero is never used as a code byte, nor does it ever appear in the data section of any code block. This means that COBS takes an input consisting of characters in the range [0,255] and produces an output consisting of characters only in the range [1,255]. Having eliminated all zero bytes from the data, a zero byte can now be used unambiguously to mark boundaries between packets. This allows the receiver to resynchronize reliably with the beginning of the next packet after an error. It also allows new listeners to join a broadcast stream at any time and reliably detect where the next packet begins.

Code	Followed by	Meaning
0x00	(not applicable)	(not allowed)
0x01	nothing	A single zero byte
0x02	one data byte	The single data byte, followed by a single zero
0x03	two data bytes	The pair of data bytes, followed by a single zero
0x04	three data bytes	The three data bytes, followed by a single zero
n	(*n*-1) data bytes	The (*n*-1) data bytes, followed by a single zero
0xFD	252 data bytes	The 252 data bytes, followed by a single zero
0xFE	253 data bytes	The 253 data bytes, followed by a single zero
0xFF	254 data bytes	The 254 data bytes, **not** followed by a zero

Table 1. Code values used by Consistent Overhead Byte Stuffing

Apart from one exception, a code byte value of *n* represents a sequence of *n* bytes that ends with a zero. The (*n*-1) non-zero bytes are placed immediately after the code byte; the final trailing zero is implicit. Note that this means that *n* bytes of code (a code byte plus (*n*-1) data bytes) are used to encode the same number of bytes — *n* bytes — from the source data, which gives zero encoding overhead. The one exception is code byte 0xFF, in which case 255 output bytes are used to encode a sequence of 254 bytes from the source data, giving an encoding overhead of $^1/_{254}$ (a little under 0.4%) in that case.

It is also simple to eliminate some value other than zero from the data stream, should that be necessary. For example, a radio interface that connects through the serial port like a Hayes modem might use an ASCII carriage return (byte value 0x0D) to mark the end of each packet [Che95]. By taking the output of COBS and XORing each byte with 0x0D before sending it to the radio, the COBS output is easily converted into a form suitable for transmission over this radio device. The only value that gives the result 0x0D when XORed with 0x0D is zero, and since there are no zeroes in the output from COBS, this procedure cannot result in any occurrences of 0x0D in the converted output. The receiver XORs each received byte with 0x0D to reverse the transformation before feeding it to the COBS decoder, so that the data is decoded correctly. Of course, in real implementations, the COBS encoding and the output conversion are performed in a single loop for efficiency reasons [Cla90].

For the remainder of this paper we will consider, without loss of generality, only the case of eliminating zeroes from the data. The process described above can be used in any case where a different value is to be eliminated.

3.2 Encoding Procedure for COBS

The job of the COBS encoder is to break the packet into one or more sequences of non-zero bytes. The encoding routine searches through the first 254 bytes of the packet looking for the first occurrence of a zero byte. If no zero is found, then a code of 0xFF is output, followed by the 254 non-zero bytes. If a zero is found, then the number of bytes examined, n, is output as the code byte, followed by the actual values of the (n-1) non-zero bytes up to (but not including) the zero byte. This process is repeated until all the bytes of the packet have been encoded.

The one minor problem with this simple description of the algorithm is that the last sequence of a packet must either end with a zero byte, or be exactly 254 bytes long. This is not true for all packets. To circumvent this problem, a zero byte is logically appended to the end of every packet before encoding; after decoding, the final zero of every packet is discarded to correctly recreate the original input data. This ensures that all packets do end with a zero and are therefore encodable. It is not necessary actually to add this zero byte to the end of the packet in memory; the encoding routine simply has to behave as if the added zero were there. As an optional optimization, packets that when encoded happen to end with a final code 0xFF block (sequence without a final zero) do not need to have a final zero logically appended. This optimization can be performed without ambiguity, because the receiver will observe that the decoded packet does not end with a zero, and hence will realize that in this case there is no trailing zero to be discarded in order to recreate the original input data. Figure 2 shows an example of packet encoding.

The implementation of COBS is very simple. The Appendix to this paper gives complete C source code listings to perform both COBS encoding and decoding. Both algorithms have running time that is linear with respect to packet size, and can be implemented in extremely simple hardware. No multiplications or divisions are required, nor are additions or subtractions. Both algorithms can be implemented using only assignment, increment, and test for equality.

One aspect of COBS that differs from conventional two-for-one substitution encodings like PPP is that PPP operates on a single byte at a time, whereas COBS has to 'look ahead' up to 254 bytes. In the context of thinking about network data as a continuous stream

Input:	45	00	00	2C	4C	79	00	00	40	06	4F	37	00
Output:	02	45	01	04	2C	4C	79	01	05	40	06	4F	37

Figure 2. Example Encoding

The input data is shown with a phantom zero logically appended to the end. The encoded form is shown below, with code bytes shaded. Note that the encoded form contains no zeroes and that it is exactly one byte longer than the input (not counting the phantom zero).

of bytes, this suggests that COBS requires more buffering and consequently adds more delay to the stream than PPP does. However, thinking about network data as a continuous stream of bytes can be misleading. While it is true that hardware usually transmits data serially, it is much more common for programming interfaces to operate a packet at a time rather than a byte at a time. When the operating system has a packet to send, it does not slice the packet into single bytes and hand them to the network device driver one at a time; it makes a single call that passes the entire packet to the driver. The driver then has access to the entire packet, which it can encode and send immediately.

3.3 Behavior of COBS

The low overhead of COBS is due to the fact that, in most cases, the size of each code block is exactly the same as the size of the data sequence it encodes. Consider a sequence of n input bytes ($n \leq 254$) that consists of (n-1) non-zero bytes, and a single zero byte at the end. That sequence is encoded using a code block containing a single code byte, and (n-1) data bytes. Hence n input bytes are encoded using exactly n output bytes, so the output size is the same as the input size.

The worst cases for PPP and for COBS occur in opposite circumstances. For PPP, the pathological case where the packet consists entirely of reserved values (0x7D or 0x7E) is the case that results in the size of the packet being doubled. For COBS, the pathological case is the opposite case — the case where the packet contains no occurrences of the reserved value (zero). In this case, each 254 bytes of packet data is encoded using 255 bytes of output data — one code byte (0xFF) and the 254 non-zero data bytes. This encoding ratio of 255 output bytes for every 254 input bytes results in an overhead of about 0.4% for large packets. For COBS, the case where the packet consists entirely of the reserved value (zero) is not a pathological case at all. Every zero byte is encoded using just a single code byte, resulting in zero overhead.

The property that COBS has at most one byte of overhead for every 254 bytes of packet data is very good for large packets, but has an unfortunate sideeffect for small packets. Every packet 254 bytes or smaller always incurs exactly one byte of overhead.

We regard one byte of overhead for small IP packets a small price to pay in exchange for the significant performance gains we get from the ability to send much larger IP packets than would otherwise be possible. However, there could be circumstances where any cost, however small, is unacceptable. To address this concern we have also experimented with a minor modification to basic COBS called Zero Pair Elimination (ZPE), which exhibits better performance for small packets, as described in the next section.

3.4 Zero Pair Elimination

In our experiments on real-world data (see Section 5) we observed that not only is zero a common value in Internet traffic, but that adjacent pairs of zeros are also very common, especially in the IP headers of small packets. To take advantage of this we shortened the maximum encodable sequence length, freeing some codes for other uses. These codes were reassigned to indicate sequences ending with a *pair* of implicit zeroes. The reduction in the maximum sequence length increases the worst-case overhead, so we do not wish to reduce the maximum length by too much. Empirically we found that reassigning 31 codes gave good performance while still maintaining a reasonable worst-case bound.

Codes 0x00 to 0xDF have the same meaning as in basic COBS, and code 0xE0 encodes the new maximum length sequence of 223 bytes without an implicit zero on the end. This gives COBS/ZPE a worst-case overhead of one byte in 223. Codes 0xE1 to 0xFF encode sequences that end with an implicit pair of zeroes, containing, respectively, 0 to 30 non-zero data bytes.

This has the good property that now some of our code blocks are *smaller* than the data they encode, which helps mitigate the one-byte overhead that COBS adds. Figure 3 shows an example of a small packet (actually the beginning of a real IPv4 packet header) that gets one byte smaller as a result of encoding using COBS/ZPE. The disadvantage of using COBS/ZPE is a slightly poorer worst-case overhead — about 0.45% instead of 0.40% — but this is still a very small worst-case overhead. In fact, as described in Section 5, when a typical mix of real-world network traffic is encoded using COBS/ZPE, it actually gets smaller by about 1%. PPP byte stuffing cannot compete with this since PPP never makes any packet smaller.

COBS/ZPE is useful because pairs of zeroes are common in packet headers. Also, the trend towards aligning packet fields on 64-bit boundaries in high-performance protocols sometimes results in padding

Input:	45	00	00	2C	4C	79	00	00	40	06	4F	37	00
Output:	E2	45	E4	2C	4C	79	05	40	06	4F	37		

Figure 3. Example Encoding with Zero-Pair Elimination

The input data is shown with a phantom zero logically appended to the end. The encoded form using COBS/ZPE is shown below, with code bytes shaded. Note that the encoded form contains no zeroes, and that it is one byte shorter than the input (not counting the phantom zero).

zeroes between fields. These padding zeroes waste precious bandwidth on slow wireless links, and using COBS/ZPE can help to mitigate this effect by encoding these patterns more efficiently.

We have avoided using more computationally expensive compression algorithms such as Huffman encoding [Huff52] [Knu85] and Lempel Ziv [LZ77] [Wel84]. Although, like PPP, they may have good average performance, for some data they can make the packet bigger instead of smaller [Hum81], and that is contrary to our goal of ensuring a tight bound on worst-case performance. We also believe it is inappropriate to attempt heavyweight compression at the link layer. The majority of compressible data is much more effectively compressed before it even reaches the IP layer using data-specific algorithms such as JPEG [ISO10918] for images and MPEG [ISO11172] for video. Finally, we expect to see more Internet traffic being encrypted in the future, particularly traffic over wireless links, and it is not possible to compress data after it has been properly encrypted. ZPE is a lightweight technique that works well for small packets that contain zero pairs, without sacrificing the benefit of a tight bound on worst-case overhead for large packets that do not.

3.5 Lower-level Framing

COBS is defined in terms of byte operations. This means that there also needs to be some underlying mechanism to detect correctly where the byte boundaries fall in the bit-stream. When used over an RS232 serial port, the start/stop bits perform this function, at a cost of 20% extra overhead. HDLC framing is usually more efficient than this, but it too, in the worst case, can add as much as 20% overhead. The benefit of COBS having such a tight bound on worst-case overhead at the byte level is somewhat diminished if the bit-level framing it depends upon is not similarly well behaved.

Fortunately, bit-level framing of COBS-encoded data can be performed with no additional variable overhead, simply by choosing an appropriate bit-level framing pattern: one that is known not to appear anywhere in COBS-encoded data. If the framing pattern cannot inadvertently appear, there is no need for any bit stuffing mechanism to eliminate it.

The choice of framing pattern is easy. Since COBS-encoded data contains no zero bytes, we know that there is at least one binary ‘1’ bit somewhere in every byte. This means that in a bit-stream of COBS-encoded data there can be no contiguous run of more than fourteen zero bits, which suggests an obvious candidate for the COBS end-of-packet marker: a run of fifteen (or more) zero bits, followed by a single binary ‘1’ bit to signal the beginning of the next packet. Used this way, COBS guarantees a very tight bound, not only on the number of bytes, but also on the total number of bits required to transmit a data packet.

Some kinds of network hardware cannot send long runs of zero bits reliably, and in these cases bit stuffing is used not only for framing purposes, but also to impose a bound on the maximum number of consecutive zero bits that can appear. Providing that the length limit imposed by the hardware is not less that fourteen zero bits, COBS encoded data can be sent reliably without requiring any further bit stuffing.

4. Theoretical Analysis

In this section we compare the best case, worst case, and average case encoding overhead, given uniformly distributed random data, for COBS and for PPP byte stuffing. It is useful to calculate the expected performance for random data, because data that is well-compressed and/or well-encrypted should have a uniform distribution of byte values. Section 5 presents the actual performance on today’s real packets, which are not all compressed and/or encrypted.

4.1 Best Case Overhead

The best case for PPP byte stuffing is a packet that contains no occurrences of the reserved (0x7D or 0x7E) characters. In this case, PPP encoding adds no overhead to the packet at all.

The best case for COBS is a packet with plenty of zeroes, so that nowhere in the packet is there any contiguous sequence of more than 254 non-zero bytes. In this case, each block is encoded with no overhead at all. Counting the phantom zero that has to be added to every packet before encoding, this results in a best-case overhead of a single byte, for any size of packet.

4.2 Worst Case Overhead

The worst case for PPP byte stuffing is a packet that consists entirely of reserved (0x7D or 0x7E) characters. In this case, encoding doubles the size of the packet, giving an overhead of 100%.

The worst case for COBS is a packet that contains no zeroes at all. In this case, each sequence of 254 bytes of packet data is encoded using 255 bytes of output data, giving one byte of overhead for every 254 bytes of packet data. For example, a maximum size IP packet over Ethernet is 1500 bytes long, and in the worst possible case COBS would add an overhead of six bytes to a packet of this size.

4.3 Expected Overhead

The best and worst cases for each algorithm are easy to calculate, but they do not give the whole picture. It would be useful also to know the overall efficiency we expect to achieve for a large number of packets, in terms of what percentage of transmitted bytes we expect to contain useful data and what percentage we expect to be encoding overhead. Since byte stuffing is a process that takes as input a packet composed of characters from an alphabet of 256 possible symbols and gives as output a packet composed of characters from an alphabet of only 255 possible symbols, in general there must be some overhead. Exactly how much longer a particular packet becomes may or may not depend on the contents of that packet, depending on the algorithm being used. With some algorithms there may be fortunate packets that incur no overhead at all, but information theory tells us that for random data the average overhead must be at least:

$$\frac{\log 256}{\log 255} - 1 \approx 0.0007063, \text{or roughly } 0.07\%$$

This theoretical bound gives us a metric against which to judge different byte stuffing schemes. Some algorithms may be able to beat this bound for some packets, but there is no algorithm that can beat this bound for all packets.

4.3.1 Expected Overhead for PPP

For PPP the expected overhead is easy to calculate. PPP has only two distinct behavior patterns: it either reads a single byte and writes a single byte, or it reads a single byte and writes a pair of bytes. In uniformly distributed random data, the probability that any given byte will be one of PPP’s two reserved values, causing PPP to output two bytes instead of one, is $^2/_{256}$. In a packet of length n, there will be on average $n \times {}^2/_{256}$ occurrences of reserved values and $n \times {}^{254}/_{256}$ occurrences of other values. This gives an expected output length of:

$$n \times 2 \times \tfrac{2}{256} \; + \; n \times 1 \times \tfrac{254}{256} \; = \; 1.0078125n$$

An expected output length 1.0078125 times the input length gives an expected overhead of 0.78125%, about 11 times worse than the theoretical optimum.

4.3.2 Expected Overhead for COBS

For COBS the average overhead is a little harder to calculate since COBS has 255 different behaviors, rather than just the two that PPP has. In addition, not only does each behavior write a different number of output bytes, each behavior also reads different numbers of input bytes. Since the number of input bytes read is not always one as it is for PPP, we must also calculate the average number of bytes read per code block, and divide the average output by the average input to determine the overall average overhead.

First we calculate the average input per code block. If the first byte the algorithm encounters is a zero, then that single byte is encoded as a code block. The probability P(1) of this happening is $^1/_{256}$. If the first byte is not a zero, but the second byte is, then the algorithm reads two bytes and outputs a block. The probability P(2) of this happening is $^{255}/_{256} \times {}^1/_{256}$. The probability P($n$) that it reads n-1 non-zeroes ($n \le 254$) followed by a zero is:

$$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256}$$

The longest code block, code 0xFF, occurs when the algorithm encounters 254 non-zeroes without seeing a single zero. The probability P(255) of this happening is $({}^{255}/_{256})^{254}$, and in this case the algorithm reads 254 bytes and outputs a block of 255 bytes.

The average input per code block is therefore:

$$\left(\sum_{n=1}^{254} n \times P(n)\right) + 254 \times P(255)$$

Now we calculate the average output per code block. The probabilities of each different behavior remain the same, and for all codes except one the number of bytes output is also exactly the same as the number of bytes input. The exception is code 0xFF, where the number of bytes input is 254 but the number of bytes output is 255.

The average output per code block is therefore:

$$\left(\sum_{n=1}^{254} n \times P(n)\right) + 255 \times P(255)$$

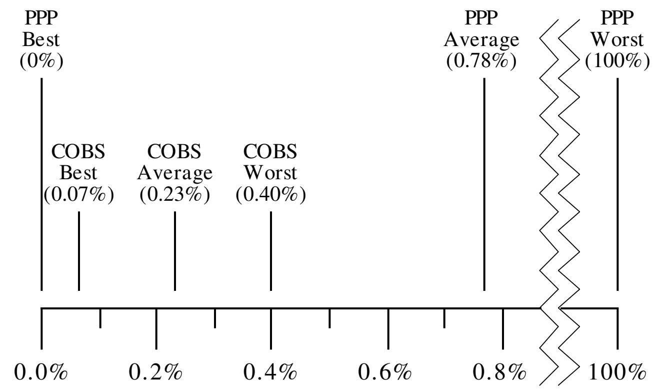

Figure 4. Encoding Overhead for 1500 Byte Packet
PPP's best, average, and worst cases vary widely. In contrast, COBS's best, average and worst cases all fall within a narrow range, and are all better than PPP's average case.

The ratio of average output divided by average input is:

$$\frac{\left(\sum_{n=1}^{254} n \times P(n)\right) + 255 \times P(255)}{\left(\sum_{n=1}^{254} n \times P(n)\right) + 254 \times P(255)} \approx 1.002295$$

The theoretical average overhead for COBS on random data is therefore a little under 0.23%, more than three times better than PPP's average.

Figure 4 summarizes these results both for PPP and for COBS. These results hold for well compressed packets which contain a uniform distribution of byte values, but not all Internet traffic is well compressed. In addition, it is not possible to have fractional bytes of overhead. In theory a 40-byte IPv4 TCP acknowledgement packet encoded with COBS may average an overhead of $40 \times 0.23\% = 0.092$ bytes, but in practice that fraction is rounded up to an entire byte of overhead. For small packets this rounding up to the next whole byte may be a more dominant contributor to overhead than the actual underlying properties of the algorithm. To investigate how much effect this potential problem might have, we encoded traces of real-world network traffic using both COBS and PPP byte stuffing, and these results are presented in the next section.

5. Experimental Results

Real-world network traffic may not behave the same way as our theoretical traffic. In real-world traffic, small packets are common, and not all data is compressed and/or encrypted. To see how these factors affect the algorithms, we gathered traces of network traffic using tcpdump [Jac89] and compared how efficiently those packets were encoded by PPP, COBS and COBS/ZPE. We present results for two of the traces here.

For the benefit of readers who wish to see how COBS performs on other traffic mixes, the analysis tool, which will read any standard format tcpdump log file, is available at the authors' Web Site <http://mosquito-net.Stanford.EDU/software/COBS>. (Note: to evaluate how efficiently COBS encodes entire packets — both header and payload — the tool should be used on logs that contain complete packets, not just packet headers. The tool will give a warning if the log contains incomplete packets.)

5.1 Three-day Trace

One of our colleagues frequently works at home, and his sole Internet connection is via a portable ISM-band packet radio attached to his laptop computer. We collected a trace of all his packets for a period of three days. We regard this trace as being representative of a user who makes extensive use of a wireless interface. The trace contains 36,744 IP packets, totalling 10,060,268 bytes of data (including IP headers and higher layers; not including the link-level header). The MTU of his wireless interface was 1024 bytes, giving a worst-case COBS overhead for large packets of five bytes.

However, most of the packets captured were not large; 69% of the packets were shorter than 254 bytes and necessarily incurred exactly one byte of overhead when encoded with COBS. Moreover, 41% of the packets were exactly 40 bytes long, which is just the length of a TCP acknowledgement containing no data. Another 10% of the packets were exactly 41 bytes long, which is the length of a TCP packet containing just one data byte. Taking these two numbers together, this means that over half the packets were 40 or 41 bytes long. Only 15% of the packets were maximum-sized 1024-byte packets.

We regard this as a particularly challenging test case with which to evaluate COBS, because it contains so many small packets. The results for this trace file are shown in Figure 5.

PPP incurred a total overhead of 36,069 bytes (0.36%). 74% of the packets incurred no overhead, but some packets incurred much more. More than 100 packets incurred 15 bytes of overhead or more, and one packet incurred as much as 53 bytes of overhead. In this trace no packets incurred more than 53 bytes of overhead, supporting the claim that real packets do not come close to the factor-of-two overhead that conventional byte stuffing forces us to design for.

COBS incurred a total overhead of 57,005 bytes (0.57%), meaning that even in this unfavourable test case COBS costs only 0.21% extra compared to PPP, for the benefit of having a tight bound on the worst-case overhead. 74% of the packets had exactly one byte of overhead, 7% had two bytes, 8% had three bytes, and 11% had four.

COBS/ZPE maintained a tight bound on worst case performance while doing on average much better than either PPP or COBS. For a 1024-byte packet the maximum possible COBS/ZPE overhead is five bytes, but in fact in this trace no packet incurred more than four. In addition COBS/ZPE *reduced* the overall size of the data by 26,238 bytes, giving a net overall saving of 0.26%.

5.2 MPEG Trace

With the increasing popularity of the World Wide Web, we expect to see large packets and compressed data (particularly image data) becoming more common on the Internet. To see how COBS would perform under these conditions we captured a large bulk transfer of compressed image data. The data file was MIRV.MPG, a 15.3MB MPEG file of an MTV music video, and it was transferred using ftp [RFC959]. The trace contains 25,858 IP packets, totalling 18,269,430 bytes of data. The MTU of the wireless interface was 1088 bytes, giving a worst-case COBS overhead for large packets of five bytes.

This trace is more favourable to COBS because it contains many large packets. 63% of the packets were maximum-sized IP packets, 1088 bytes long. The results for this trace file are shown in Figure 6.

PPP incurred a total overhead of 101,024 bytes (0.55%). 36% of the packets incurred no overhead, but most incurred much more. The majority of packets incurred 1-10 bytes of overhead and one packet incurred as much as 20 bytes of overhead. In this trace no packets incurred more than 20 bytes of overhead, supporting the claim that real packets do not come close to the factor-of-two overhead that conventional byte stuffing forces us to design for.

COBS incurred a total overhead of only 35,410 bytes (0.19%), beating PPP. 77% of the packets had exactly one byte of overhead. Only 17 packets in the entire trace incurred five bytes of overhead and as expected, no packets incurred more than that.

COBS/ZPE maintained a tight bound on worst case performance while doing on average much better than either PPP or COBS. For a 1088-byte packet the maximum possible COBS/ZPE overhead is five bytes, but in fact in this trace no packet incurred more than four bytes of overhead. In addition COBS/ZPE *reduced* the overall size of the data by 161,548 bytes, giving a net overall saving of 0.88%.

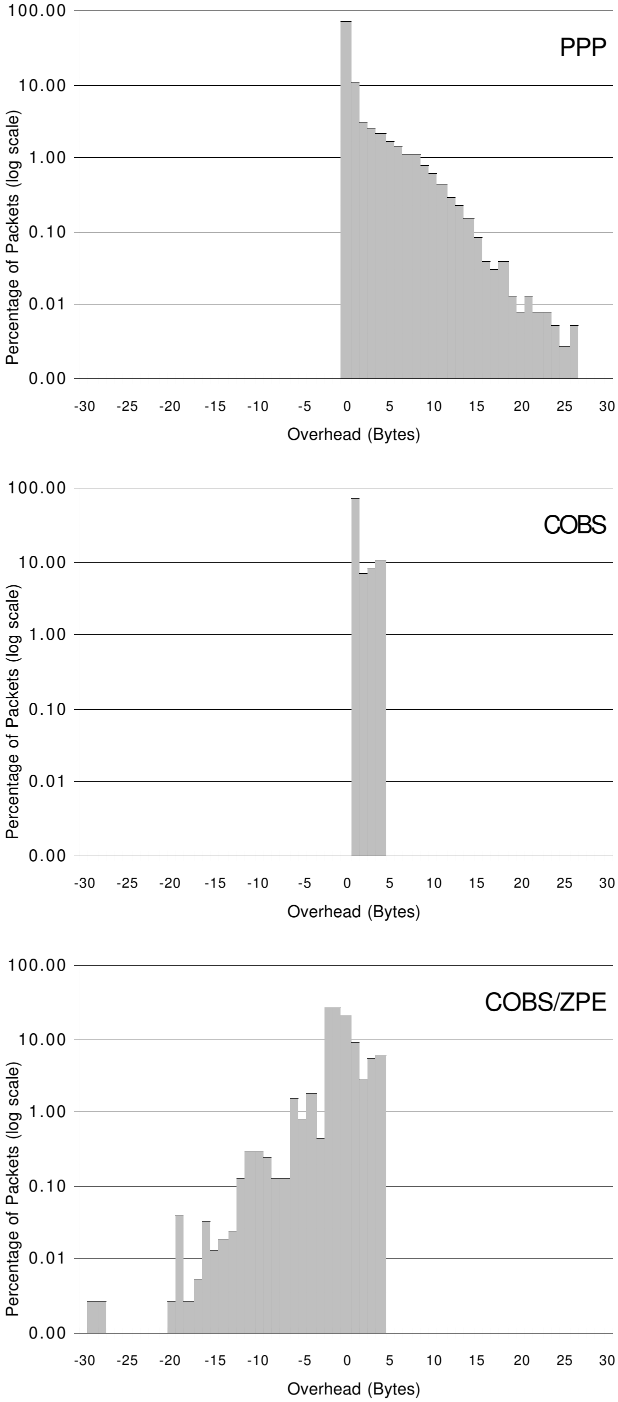

Figure 5. Three-day Trace

Histograms showing, for each amount of overhead indicated on the horizontal axis, the percentage of packets that incur that overhead. For our traces 0.001% of the packets is less than one packet, and since the only number of packets less than one is zero packets, we chose 0.001% as the base line of the log scale. For PPP the lowest overhead is zero bytes and increases up to a maximum of 53 bytes (far off the scale to the right). For COBS the overhead is concentrated in a tight spike in the middle: every packet incurred 1-4 bytes of overhead. For COBS/ZPE the overhead is at most four bytes; most packets actually incurred negative overhead (net compression). A few especially compressible packets incurred a negative overhead of more than 300 bytes (far off the scale to the left).

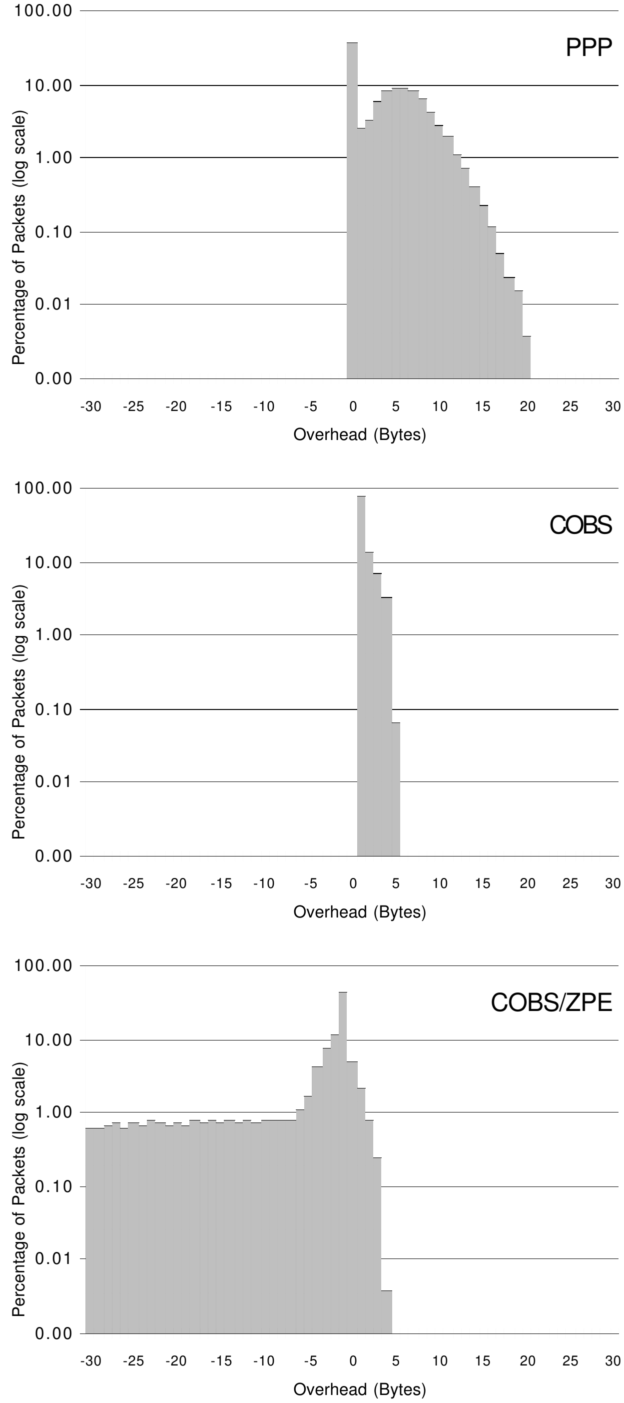

Figure 6. MPEG Trace

Histograms showing, for each amount of overhead indicated on the horizontal axis, the percentage of packets that incur that overhead. For our traces 0.001% of the packets is less than one packet, and since the only number of packets less than one is zero packets, we chose 0.001% as the base line of the log scale. For PPP the overhead begins at zero and increases up to a maximum of 20 bytes. For COBS the overhead is concentrated in a tight spike in the middle: every packet incurred between one and five bytes of overhead. For COBS/ZPE the overhead is at most four bytes; most packets actually incurred negative overhead (net compression) and in fact the histogram continues off the scale far to the left: A few especially compressible packets incurred a negative overhead of more than 500 bytes.

6. Conclusions

COBS is a useful addition to our arsenal of techniques for data communications. It is simple to understand, easy to implement, and gives significant performance benefits by allowing much larger packets to be sent over a given piece of network hardware.

COBS is easy to implement efficiently in software, even for primitive microprocessors. In one project in our group [Pog96] COBS has been implemented in hand-written eight-bit assembly code to allow a small embedded control device to connect to a wireless interface and communicate with Internet hosts using UDP/IP. The device needs to be able to send and receive 1K blocks of data but does not have enough memory for either an implementation of TCP or of IP fragmentation and reassembly. Without COBS it would have been much harder to make the device work. We would have had to add extra memory and would have had to do a lot of extra development work to implement TCP and/or IP fragmentation and reassembly in eight -bit assembly code.

In retrospect it is surprising that COBS or similar techniques have never been described in the literature before. Perhaps one reason is that, until the development of unlicensed radio transmitters under the FCC's Part 15 ISM band rules, the networking community had not confronted the problem of dealing with devices where the maximum transmission size is a hard limit dictated at the physical level.

We see no reason why COBS should not become the algorithm of choice, not only for packet radio applications, but for all future software that uses byte stuffing. In principle existing software could also be rewritten to use COBS, but in practice market inertia makes this unlikely. For software that currently works acceptably, such as PPP over telephone modems, there is little incentive to change to a new encoding, but for all new software that requires byte stuffing we believe that COBS should be given serious consideration.

The benefit of conventional two-for-one substution encodings like PPP, compared to COBS, is that they may encode small packets with no overhead whereas basic COBS always adds exactly one byte. However, three factors make this apparent benefit of conventional byte stuffing algorithms less compelling.

The main factor is that the pressing problem for many new wireless devices is that of sending large packets, not small packets. It is the large packets that cause problems, because the software must be able to ensure that they do not exceed the device's physical and regulatory limits.

Another factor is that the move to IPv6 [RFC1883] in the future means that the very smallest packets, where PPP does better than COBS, will become increasingly uncommon. Although we expect to see header compression techniques being used to reduce the overhead of the IPv6 header (especially over slow links), those header compression techniques will reduce the header size by amounts measured in tens of bytes, dwarfing concerns about differences of a single byte here and there.

Finally, if even a single byte of overhead is unacceptable, a trivial modification to COBS to support Zero Pair Elimination makes it perform better than PPP, even for short packets. COBS/ZPE beats both PPP's average overhead and its worst case overhead.

Further information about COBS is available in [Che97].

7. Acknowledgements

We are grateful for the helpful comments on this paper from Craig Partridge, Diane Tang, Xinhua Zhao, Petros Maniatis, Mema Roussopoulos, Elliot Poger, Kevin Lai, Akihiro Tominaga, Jonathan Stone, Hugh Holbrook, Brendon Whateley, Tom Costello, Anna Patterson, Neil Crellin, Edouard Bugnion, Eric Hawkes, Maynard Handley, Ann McLaughlin, Richard Ford, Charlie Payne, Nicholas Tingle and the anonymous SIGCOMM reviewers.

We also thank Hugh Holbrook for making traces of his wireless traffic available to us, including the three-day trace presented in this paper.

This work was supported by Rockwell International Science Center, FX Palo Alto Laboratory, AT&T/Lucent and NSF Faculty Career grant number CCR-9501799.

8. References

[ARRL84] AX.25 Amateur Packet-Radio Link-Layer Protocol Version 2.0, October 1984. Available from the American Radio Relay League, Newington CT USA 06111, and other sources.

[CA-96.26] CERT℠ Advisory CA-96.26. Denial-of-Service Attack via ping, December 1996.

[Che95] Stuart Cheshire and Mary Baker. Experiences with a Wireless Network in MosquitoNet. *IEEE Micro*, February 1996. An earlier version of this paper appeared in *Proceedings of the IEEE Hot Interconnects Symposium '95*, August 1995.

[Che97] Stuart Cheshire. *Consistent Overhead Byte Stuffing*, Ph.D. Thesis, Computer Science Department, Stanford, 1997.

[Cla90] David Clark and David Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Proceedings of ACM SIGCOMM 1990,* September 1990.

[ECMA40] European Computer Manufacturers Association Standard ECMA-40: HDLC Frame Structure.

[Hum81] Pierre Humblet. Generalization of Huffman Coding to Minimize the Probability of Buffer Overflow. *IEEE Transactions on Information Theory*, Vol. IT-27, pp. 230-232, March 1981.

[Huff52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE,* Vol.40, No.9, September 1952, pp.1098-1101.

[IEEE802.5] Token Ring Access Method and Physical Layer Specifications. Institute of Electrical and Electronic Engineers, IEEE Standard 802.5-1989, 1989.

[ISO10918] ISO Committee Draft 10918. Digital compression and coding of continuous-tone still images, ISO/IEC 10918, 1991.

[ISO11172] ISO Committee Draft 11172. Information Technology-Coding of moving pictures and associated audio for digital storage media up to about 1.5 Mbit/s, ISO/IEC 11172-1, 1993.

[Jac89] V. Jacobson, C. Leres, and S. McCanne. *tcpdump,* <ftp://ftp.ee.lbl.gov/tcpdump.tar.Z>, June 1989.

[Knu85] D. E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms,* Vol 6, pp 163-180, 1985.

[LZ77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory,* May 1977.

[Pog96] Elliot Poger. Radioscope, <http://mosquitonet.Stanford.EDU/~elliot/RadioScope/>, December 1996.

[RFC959] J. Postel, J. Reynolds. File Transfer Protocol (FTP). RFC 959, October 1985.

[RFC1055] J. Romkey. A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP. RFC 1055, June 1988.

[RFC1135] J. Reynolds. The Helminthiasis (infestation with parasitic worms) of the Internet. RFC 1135, December 1989.

[RFC1662] William Simpson. PPP in HDLC-like Framing. RFC 1662, July 1994.

[RFC1883] Steve Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883, December 1995.

[US94-15] United States Title 47 Code of Federal Regulations (CFR), Federal Communications Commission (FCC) Part 15, Low-Power Device Regulations, Section 15.247. U.S. Government Printing Office.

[Wel84] T. Welch. A Technique for High-Performance Data Compression. *Computer,* June 1984.

Appendix

Source Code Listings

```
/*
 * StuffData byte stuffs "length" bytes of
 * data at the location pointed to by "ptr",
 * writing the output to the location pointed
 * to by "dst".
 */

#define FinishBlock(X) \
    (*code_ptr = (X),  \
    code_ptr = dst++,  \
    code = 0x01)

void StuffData(const unsigned char *ptr,
unsigned long length, unsigned char *dst)
    {
    const unsigned char *end = ptr + length;
    unsigned char *code_ptr = dst++;
    unsigned char code = 0x01;

    while (ptr < end)
        {
        if (*ptr == 0) FinishBlock(code);
        else
            {
            *dst++ = *ptr;
            code++;
            if (code == 0xFF) FinishBlock(code);
            }
        ptr++;
        }
    FinishBlock(code);
    }
```

Listing 1. COBS Encoding in C

```
/*
 * UnStuffData decodes "length" bytes of
 * data at the location pointed to by "ptr",
 * writing the output to the location pointed
 * to by "dst".
 */

void UnStuffData(const unsigned char *ptr,
unsigned long length, unsigned char *dst)
    {
    const unsigned char *end = ptr + length;
    while (ptr < end)
        {
        int i, code = *ptr++;
        for (i=1; i<code; i++) *dst++ = *ptr++;
        if (code < 0xFF) *dst++ = 0;
        }
    }
```

Listing 2. COBS Decoding in C