CONSISTENT OVERHEAD
BYTE STUFFING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stuart David Cheshire

March 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mary Baker (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Craig Partridge

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown

Approved for the University Committee on Graduate Studies:

Abstract

Byte stuffing is a process that encodes a sequence of data bytes that may contain ‘illegal’ or ‘reserved’ values, using a potentially longer sequence that contains no occurrences of these values. The extra length is referred to as the *overhead* of the encoding.

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

However, there are some increasingly popular network devices whose performance is determined more by the worst case than by the average case. For example, the transmission time for ISM-band packet radio transmitters is strictly limited by FCC regulation. To adhere to this regulation, the current 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 dissertation presents a new byte stuffing algorithm, called Consistent Overhead Byte Stuffing (COBS), which tightly bounds the worst-case overhead. It guarantees in the worst case to add no more than one byte in 254 to any packet. For large packets this means that their encoded size is no more than 100.4% of their pre-encoding size. This is much better than the 200% worst-case bound that is common for many byte stuffing algorithms, and is close to the information-theoretic limit of about 100.07%.

Furthermore, the algorithm is computationally cheap, and its average overhead is very competitive with that of existing algorithms.

Acknowledgements

Thanks to my advisor, Mary Baker, for all her help developing the ideas in this dissertation, for teaching me how to present my work clearly and convincingly in written form and consequently achieve a series of successful conference publications, and for her help with all those innumerable educational steps a Ph.D. student has to complete in order to graduate. Without Mary's support I would never have had the opportunity to pursue my interest in wireless networking which led to the results presented in this dissertation.

Thanks to my mentors during my years spent at Stanford University: David Cheriton, Craig Partridge and Mary Baker. Without their excellent teaching, both in the classroom and in private conversation, I may never have developed the thorough understanding, reasoned opinions, and sound intuitions necessary to produce a Ph.D. thesis in the field of computer networking.

Thanks to my Oral Defence Committee, Mary Baker, Craig Partridge, Nick McKeown, Vaughan Pratt, and John Gill, for their valuable advice and suggestions about my thesis.

Thanks to my Reading Committee, Mary Baker, Craig Partridge and Nick McKeown, for their great efforts helping me produce this dissertation, and their vigilance in helping ensure that it met the high standards they expect.

Thanks to Steven Gass, Engineering Library Head Librarian, for his assistance in searching the computing and communications literature all the way back to 1968.

Thanks to John Gill for sharing with me his 1991 design for a protocol for sending XWindow protocol units over a modem connection, and for his invaluable assistance analysing and discussing with me the performance characteristics and probabilistic behaviour of consistent overhead byte stuffing.

Thanks to Tom Costello and Anna Patterson for their assistance proof-reading this dissertation.

Thanks to my wife, Pavani Diwanji, for her emotional support through the long and sometimes difficult process of completing a Ph.D.

Thanks to Digital Equipment Corporation for creating the Alta Vista search engine, which proved to be an invaluable time-saver in locating publications and other information on the Internet.

This work was supported by:

Rockwell International Science Center

FX Palo Alto Laboratory

AT&T/Lucent

NSF Faculty Career grant number CCR-9501799

A grant from the Keio Research Institute at SFC, Keio University
and the Information-Technology Promotion Agency of Japan, and

An Alfred P. Sloan Faculty Research Fellowship

Contents

List of Tables

List of Figures

Chapter One

Introduction

This chapter explains why it is undesirable for a data stuffing algorithm to have highly variable overhead, lists the major contributions of this Ph.D. thesis, and presents the outline of the dissertation.

1.1 Problem Statement

The purpose of byte stuffing is to convert data packets into a form suitable for transmission over a serial medium like a telephone line. When packets are sent over a serial medium there needs to be some way to tell where one packet ends and the next begins, and this is typically done by using a special reserved value to indicate packet boundaries. 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 *overhead* (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 Serial Line IP (SLIP) [RFC1055], the Point-to-Point Protocol (PPP) [RFC1662] and AX.25 (Amateur Packet Radio) [ARRL84], have high variance in per-packet overhead. For example, using conventional PPP byte stuffing, the total overhead (averaged over a large number of packets) is typically 1% or less, but this overhead is not consistent for all packets. Some packets incur no overhead at all, while others incur much more; in principle some could increase in size by as much as 100%. High-level Data Link Control (HDLC) uses a bit stuffing technique [ECMA-40] that has a worst-case expansion of only 20%, but the drawback of bit stuffing is that algorithms that operate bit-by-bit are often harder to implement efficiently in software than algorithms that operate byte-by-byte.

While large variability in overhead may add jitter and unpredictability to network behaviour, it has only minor implications for running PPP over a telephone modem. An Internet Protocol host (IP host) [RFC791] using standard PPP encapsulation [RFC1662] need only make its transmit and receive buffers twice as large 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 is required 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 aware of packets as distinct entities and consequently impose some finite limit on the maximum packet size they can support. Channel-hopping packet radios that operate in the unlicensed ISM (Industrial/Scientific/Medical) bands under the FCC Part 15 rules [US94-15] are constrained by a maximum transmission time that they may not exceed. If the overhead of byte stuffing unexpectedly doubles the size of a packet, it could result in a packet that is too large to transmit legally. Using conventional PPP encoding [RFC1662], the only way to be certain that no packets will exceed the legal limit is to set the IP maximum transmission unit (MTU) 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.

Setting the MTU to only half the size that the underlying medium is capable of can significantly degrade the performance seen by the end-user. Using small packets is generally less efficient than using large packets. This is because there is some overhead associated with every packet sent, in terms of: (i) extra data (header bytes) transmitted over the medium, (ii) CPU cycles expended at the sender and receiver, and (iii) other additional time delays that may be incurred depending on the medium in use. For example, for every packet sent over a Metricom packet radio, some time is expended in tuning the transmitting and receiving circuitry and in exchanging preliminary handshaking information before the actual packet transmission begins [Che96]. Using Metricom packet radios, the time to send a packet is roughly 50ms of fixed overhead, plus 0.1ms per byte of packet data. Thus the time it takes to send a 1000-byte packet containing a 40 byte header and 960 bytes of user data is roughly 50ms + 100ms = 150ms. Metricom radios are half-duplex (they cannot transmit and receive simultaneously) so, assuming one 40-byte Transmission Control Protocol (TCP) acknowledgement for every two data packets [RFC1122], it takes 150ms + 150ms + 54ms = 354ms to transfer 1920 bytes of user data, giving an end-user throughput of 5424 bytes per second. If we halve the MTU to 500 bytes (a 40 byte header and 460 bytes of user data) then it takes 100ms + 100ms + 54ms = 254ms to transfer 920 bytes of user data, giving an end-user throughput of 3622 bytes per second. In this case halving the MTU unnecessarily reduces the maximum throughput by roughly 33%, which is a significant performance penalty.

Although, as seen from experiments presented in this dissertation, 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 array-bounds errors such as the one in the Unix 'finger' dæmon exploited by the famous Internet Worm of 1989 [RFC1135].

One way to avoid the performance degradation of halving the MTU might be to set the IP MTU closer to the underlying device MTU, and then use some kind of packet fragmentation and reassembly to handle those packets that when encoded become too large to send. One kind of packet fragmentation is link-layer fragmentation ('transparent fragmentation'). While this could work, requiring link-layer software to perform fragmentation and reassembly is a substantial burden to impose on driver writers. In [Kent87], Kent and Mogul wrote "… transparent fragmentation has many drawbacks … gateway implementations become more complex and require much more buffer memory …" Fragmentation and reassembly provides more opportunities for software bugs [CA-96.26] and adds protocol overhead, because the link-layer packet headers have to contain additional fields to support the detection and reassembly of fragments at the receiving end.

A second way to use fragmentation would be to avoid adding a new layer of fragmentation and reassembly at the device level, and instead to use IP's own existing fragmentation and reassembly mechanisms, but this 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. If, after encoding by the device driver, a particular packet becomes too large to send, there is no mechanism for the driver to hand the packet back to IP with a message saying, "Sorry, I failed, can you please refragment this packet into smaller pieces and try again?" Also, some software goes to great lengths to select an optimal packet size. It would be difficult for such software to cope with an IP implementation where the MTU of a device varies for every packet. Finally, it is unclear, if we were to create such a mechanism, whether it should respect the IP header's 'Don't Fragment' bit. Ignoring this bit violates the semantics that other Internet hosts expect, but obeying it and rejecting oversized packets could result in an infinite-loop livelock situation. Another host may send an unlucky packet which significantly increases in size and is rejected, but when it performs a new path MTU discovery [RFC1191] using a probe packet containing some different pattern of data, it finds that the packet size is apparently within the acceptable MTU, so it sends the unlucky packet again, and has it rejected again, and so on, potentially without end.

Given all these problems, we would prefer to use a byte stuffing algorithm that does not have such inconsistent behaviour. This dissertation 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 computationally cheap, easy to implement in software, and has a bound on its worst-case performance that is better even than HDLC's bit stuffing scheme. 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 is therefore roughly 0.4% of the packet size, rounded up to a whole number of bytes.

Using Consistent Overhead Byte Stuffing, the IP MTU may be set as high as 99.6% of the underlying device's maximum packet size without fear of any packet inadvertently exceeding that limit. Thus COBS provides better end-user performance than existing mechanisms, because with COBS a given piece of hardware can consistently send larger IP packet payloads.

All the data stuffing algorithms discussed in this dissertation, including COBS, have execution times that are O(n) with respect to the size of the data set being stuffed. The constant factor varies depending on the algorithm and the implementation, but all the algorithms execute in linear time.

1.2 Primary Contributions of this Ph.D. Thesis

The major contributions presented in this dissertation are:

- A new algorithm for encoding packet data.

- A new way of thinking about link-layer data encoding algorithms:
  - As well as the average case, the worst case also matters.
  - The problem of worst-case overhead is solvable. Having good average performance and a low bound on worst-case performance are not incompatible goals. It is possible to achieve both simultaneously.

- An explicit enumeration of the properties required of a framing protocol, and a thorough study of how existing framing protocols compare in terms of these properties.

- A new class of encoding algorithms which maintain good worst-case bounds. This dissertation explicitly presents basic COBS plus one variant called COBS/ZPE, but these are just two examples of the COBS technique. By assigning different meanings to some of the code bytes it is easy to create additional variants that have slightly different properties, as appropriate for the intended usage, without sacrificing the guarantee of low worst-case overhead.

1.3 Outline of Dissertation

The remainder of this dissertation proceeds as follows:

Chapter 2 gives more background on the problems of packet framing and data-transparency, and briefly reviews historical approaches to solving these problems. It identifies a set of desirable properties of framing schemes: Transparency, Reliability and Locality, and shows how most existing framing schemes fail to meet these three requirements. Furthermore, of the few current protocols that do meet the requirements, none can guarantee to do so efficiently for all packets.

Chapter 3 describes my new framing protocol called Consistent Overhead Byte Stuffing (COBS). COBS meets all of the requirements of a good framing protocol set out in Chapter 2, while at the same time guaranteeing that for all packets, no matter what their contents, the overhead will be small (no more than one byte of overhead for every 254 bytes of packet data). In addition to the basic algorithm, Chapter 3 also presents a simple variant called Zero Pair Elimination (COBS/ZPE) which improves on straight COBS's average performance at the expense of a fractionally higher worst-case bound.

Chapters 4 and 5 compare the costs of HDLC, conventional PPP, and COBS for encoding the same data packets. In the least favourable case for COBS, network traffic consisting predominantly of small packets, COBS is found to add less than 0.5% additional average overhead compared to conventional 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. COBS/ZPE achieves an average overhead lower than PPP's, even for small-packet traffic, without giving up COBS's advantage of also guaranteeing a very low bound on worst-case overhead. Chapter 4 considers the expected overhead from a theoretical point of view, for data consisting of uniformly distributed random eight-bit values. Real network traffic often does not have a uniform distribution of byte values, so Chapter 5 presents experimental results comparing COBS with conventional PPP for real network traffic.

Chapter 6 presents four extensions to the basic consistent overhead byte stuffing concept, and Chapter 7 presents a list of some of the applications that could benefit from COBS. Although wireless communication was the initial motivating factor in developing consistent overhead byte stuffing, the encoding that COBS performs is not inherently related to wireless technologies. There are other uses for COBS encoding, both inside and outside the field of communications systems, and this chapter describes some of those possible future uses for COBS algorithms.

Chapter 8 presents conclusions.

Chapter Two

Prior Work

This chapter gives background on the problems of packet framing and data transparency, and presents previous approaches to solving these problems.

It begins by introducing the concepts of packet framing and data stuffing and then presents a quick history of the development of long-distance communication protocols before the advent of computers. It explains how these early protocols were inadequate for today's needs, and lists the desirable properties that we would like to see in a protocol today. It explains how framing protocols are used simultaneously at many levels in a communications hierarchy and then presents brief descriptions of current computer framing protocols, grouped roughly according to their mode of operation.

2.1 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 demarcation is usually done by using a marker of some kind, 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 has indeed detected a boundary between packets. There are two basic methods for producing the special value or signal used to mark boundaries, 'out-of-band' signalling and 'in-band' signalling. A third possibility is not to use any boundary marker at all, and instead provide a field in each packet that indicates the packet's length (or mandate one particular fixed length for all packets), but this technique has a number of problems that limit its applicability, which are discussed below.

An 'out-of-band' signal is a special unique signal, such as a different physical voltage, that is outside the range of signals that payload data can generate. In low-level hardware systems 'out-of-band' framing protocols are common. For example, in IBM Token Ring the electrical signalling on the wire uses Differential Manchester Encoding [Sta94], and packet boundaries are marked by a violation of these 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 the framing signal on the cable. This approach is not a universal solution though, 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. It is also not a universal solution because in general, an out-of-band framing protocol takes a medium that is capable of transmitting n distinct signals, reserves one or more of those signals for use as a framing marker, and allows the higher-level clients use of only m distinct values for communication, where $m<n$. In this sense, out-of-band framing is a destructive process, in that it reduces the alphabet of symbols available to the clients using it by at least one, and the size of the alphabet limits the number of layers of out-of-band framing that can be used in a single communications system. Using out-of-band signalling also makes inefficient use of the communication capacity of the underlying medium; it wastes a signal that otherwise could be used to communicate packet data itself, instead of just signalling (relatively infrequent) packet boundaries.

The second way of using a special value or signal to mark boundaries is an 'in-band' signal. This is a signal that could normally occur within payload data, but one takes additional steps to ensure that it does not. One way to ensure that the signal does not appear in the data is simply to prohibit that value from being used in the data of any packet. For example, in the original version of Microsoft DOS, text files were prohibited from containing the ASCII code 0x1A, because DOS used that ASCII code to indicate 'end-of-file' [King83] [Wolv84]. This approach has the disadvantage that the communication protocol is no longer 'transparent' to the higher layers — the software using the protocol has to be aware of which character values are permitted and which are not. Non-transparent in-band framing is similar to out-of-band framing in that it reduces the alphabet of symbols available to the clients using it. The difference is that in out-of-band framing the commandeered symbols are not even in the client's alphabet of symbols, so it has no way even to refer to the special values. In non-transparent in-band framing, the commandeered symbols are actually in the client's alphabet of symbols, but the specification of the software interface prohibits their use, and an attempt by the client to use those values is considered an error.

A more useful way of implementing in-band signalling is to allow the data to contain all possible values, but to make the framing protocol software responsible for transforming any data into a form that contains no reserved values and is suitable for transmission. Whatever transformation the software 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. This kind of transmission is sometimes called 'transparent' transmission because any data values may be transmitted, without concern for the underlying transmission protocol. This dissertation concentrates on these 'transparent' transmission methods because they place no restrictions on the byte values or data patterns that may be transmitted and therefore have the widest applicability. They use the same alphabet of symbols for both their input and output, so they are non-destructive. They may be used at any level in a communications hierarchy, and any number of layers of transparent in-band framing may be used together in a single system without exhausting the number of symbols available. The mechanisms generally used to perform the data transformation necessary to enable transparent communication with in-band signalling 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). Two well-known examples of these techniques are HDLC bit stuffing [ECMA-40] and PPP byte stuffing [RFC1662]. 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 (in the worst case for PPP) doubling the packet size.

There are also 'transparent non-stuffing in-band' protocols that permit the client to use any byte values, including the values used by the protocol itself as framing markers, but perform no data stuffing operation to eliminate these values when they appear. These framing protocols work well until an error occurs, but after an error the presence of framing values within the client data may impede the resynchronization process, perhaps indefinitely.

The third possibility mentioned above is to use a length indication alone, without any end-of-frame marker value. In this case each packet is prefixed with a length value (e.g. one or two bytes) that tells the receiver how many bits or bytes of following data to expect. This technique sounds attractive but has the disadvantage that, if used alone as the sole framing mechanism, it is not guaranteed to resynchronize after an error. For example, if the length value is corrupted or lost, then the receiver will misinterpret some other byte(s) of the packet payload as being the length, and will then read the wrong number of bytes of data for the packet payload. After this the next bytes read will again most probably not be the correct length prefix for the next packet, and the problem continues. Hence a single error results in an unbounded period before another packet is correctly received, and an unbounded number of entirely correct packets may be lost due to the receiver's inability to decode them properly. The receiver may resynchronize by chance, but there is no guarantee that this will happen quickly, and indeed no guarantee that it will happen at all. For similar reasons, there is no reliable way for a receiver who joins a data-stream mid-flow to detect where the next packet boundary falls. Because of these kinds of problems, it is generally preferable to use a stateless method of detecting packet boundaries — using a special reserved value that reliably indicates a packet boundary regardless of context — rather than using a stateful method that depends on having reliable knowledge of where the previous packet ended.

ASN.1 [ISO 8824] [ISO 8825] is an example of a length- or count-based protocol. Used alone it would suffer the resynchronization problem described above, but in practice it is not used alone. ASN.1 is used to encode the data fields that make up a packet, but some other, stateless, framing protocol is used to determine where each packet begins and ends. After an error, ASN.1 may not resynchronize until the stateless framing protocol detects the beginning of the next packet and resets the ASN.1 decoder. In practice this is not a problem, since decoding partial packets is usually not useful. Thus for independent units of data, such as separate packets, a stateless framing protocol is preferable. For related units of data, such as data fields within a single packet, where it is not especially useful to receive some without the others, a stateful count-based protocol is acceptable.

2.2 Terminology

This dissertation uses the term 'alphabet' to refer to the set of primitive symbols that are available for communication. The 'vocabulary' of a protocol is the set of sentences that can be constructed from zero or more symbols in the alphabet. This dissertation uses the term 'signal' to refer to the broader signalling properties of the medium that may fall outside those used for the usual communications alphabet. For example a large negative voltage outside the normal allowable range is an electrical signal that the wire can carry, but it is not used in any valid encoding of any symbol from the normal communication alphabet.

2.3 Background: Early History

This section charts the evolution of communication techniques, from parallel communication channels, to serial communication channels with out-of-band framing, to serial communication channels with unreliable in-band framing, and finally the development of reliable in-band serial framing protocols.

2.3.1 Fire Beacons and Optical Telegraphs

Since the beginning of recorded history, human civilizations have sought to solve the problem of long distance communication. Gerard Holzmann catalogs these early attempts in *The Early History of Data Networks* [Hol95]. Early communications systems used runners, horse riders, and homing pigeons to carry written messages, and even chains of men spaced at intervals shouting to each other to convey a message down the length of the chain. To communicate more quickly over longer distances, messages were conveyed using light instead of sound or physical movement of the message. There was signalling by fire beacons on hilltops, by flashes of sunlight reflected off polished mirrors, and by men waving coloured flags or burning torches, but these early systems were, in the words of Gerard Holzmann, "somewhat ad hoc, being primarily used to send a single predefined alarm, victory, or attack signal."

The invention of the telescope in the seventeenth century enabled advances in optical communication techniques. In 1793 Claude Chappe built his semaphore telegraph in France. It used two movable bars ('indicators') attached to the ends of a larger bar (the 'regulator'). By placing the three bars in various prescribed orientations, 98 different symbols could be communicated. Some of the symbols communicated message data, and some were reserved for control information, so this system used out-of-band control signalling. A year later Abraham Edelcrantz built his shutter telegraph in Sweden. Edelcrantz's telegraph used ten shutters that could be either open or closed, forming a ten-bit binary communication path. Like Chappe, Edelcrantz defined some codes to carry message data and some to communicate control information, so this system also used out-of-band control signalling. In both of these schemes each symbol carried multiple bits of information, making them more like what today would be called a 'parallel' interface, as opposed to a 'bitwise serial' interface. Having a large number of basic symbols, instead of just the two of a binary serial interface, made it easy and relatively inexpensive to reserve some symbols for control signals.

2.3.2 Morse Code

The next major breakthrough was the development of Morse Code in the mid-nineteenth century. Morse Code allowed communication over a single electrical wire, a serial medium, in contrast to the effectively parallel visual medium used by Chappe and Edelcrantz. Morse operators still had to be able to identify boundaries between letters, between words, and between sentences, but Morse's telegraph wire had only two states — on and off — so there were no values to spare for framing information. Morse had to confront the problem of message framing a different way. Working within the constraint of the wire's two states, Morse essentially used the time domain to provide an out-of-band framing mechanism, using pauses of various lengths between the groups of dots and dashes to indicate these boundaries. In [Vail1845] Samuel Morse's business partner Alfred Vail described it thus:

> "A single touch of the key, answers to a single dot on the paper of the register; which represents the letter, E. One touch of the key prolonged, that is, the contact at the key continued for about the time required to make two dots, produces a short line, and represents T. A single touch for about the time required to make four dots, is a long line, and represents L. A single touch for about the time required to make six dots, is a still longer line and represents the 0 of the numerals. If the use of the key be suspended for about the time required to make three dots, it is a short space, used between letters. If suspended for the time required to make six dots, it is a long space, used between words, and a longer space is that used between sentences."

While Morse's telegraphic encoding scheme was a major step forward from fire beacons and optical telegraphs, it still had shortcomings. Morse did use a digital rather than analogue encoding — there are a small number of discrete allowable lengths for mark (dot or dash) and space (pause) — but Morse Code was designed for use by human operators, not computer equipment, so there was little incentive for Morse to take the next step of further constraining the number of states still further to make a binary (two-state) digital signalling system. While Morse Code did have a protocol for signalling boundaries, it was not a reliable, unambiguous, protocol. In addition to variable length dashes, the original Morse Code also used variable length pauses within letters. For example, the letter 'E' is represented by a single dot. The letter 'I' is represented by two dots separated by a short pause (the length of one dot). The letter ‘O’ is called a ‘spaced letter’ and is represented by two dots separated by a longer pause (the length of three dots). Note the ambiguity between the encoding of the letter pair ‘EE’ (dot-pause-dot) and the single letter ‘O’ (dot-pause-dot). Problems resulting from this ambiguity are covered in the ‘Practical Notes to Beginners on Sending and Receiving by the Morse Code’ in *American Telegraphy* [Mav03]:

> In words containing two or more spaced letters, space just enough, not so much as between letters. Still, do not underspace them. Avoid, for instance, making “meet her,” “mother,” or vice versa.

Modern Morse Code [Brit405-97] is a definite improvement over Morse’s original. It uses only two mark lengths, short (‘dot’) and long (‘dash’), and has no ‘spaced letters’ to cause confusion. At the turn of the century it was just one of several competing ‘Morse Codes’ (William Maver, Jr. refers to it in [Mav03] as the ‘Continental Code’), all of which marked letter boundaries using long pauses which could never appear within the encoding of any letter, and marked word boundaries using even longer pauses which could never appear within the encoding of any word. Thus Morse Code messages have a two-level hierarchical structure, using out-of-band framing at both levels. At the lower level out-of-band framing is used to denote letter boundaries, and at the higher level a different signal, that is in turn out-of-band with respect to word encoding, is used to denote word boundaries.

2.3.3 The Cost of Out-Of-Band Framing

In [Sch80] Robert Scholtz discusses why using an out-of-band signal is inefficient. It may seem to cost us nothing, but the framing marker is one member of the finite alphabet of signals that the medium is able to communicate. By removing that signal from service and reserving it solely for use as a boundary marker, the rate at which the medium can communicate information is reduced.

2.3.4 Choice of In-Band Framing Marker

The alternative to out-of-band signalling is in-band signalling. Chapter One described simple in-band signalling in the context of a medium which has a large alphabet of symbols, typically byte values in the range [0,255], where it is feasible to reserve one single value for use as the framing marker. In general though, the framing marker may be some characteristic pattern of symbols, not just a single symbol alone, because the alphabet of symbols the medium provides may not be large enough to spare a symbol for use just as a framing marker. If the medium has only two symbols (zero and one), as is common with computer communication channels, then it is definitely not possible to spare one symbol for use as a framing marker because that would leave only one symbol for communication of the packet payload, and one symbol is not enough for effective communication. In this case, the framing marker ('message prefix') has to be some characteristic sequence of binary digits. By not removing any symbols from the communications alphabet we leave the full alphabet available for data communication, but this raises the issue that now there is no automatic protection against mis-framing. The data could accidentally contain a sequence that happens to be identical to the pattern used as the message prefix, which could confuse the receiver. The longer the message prefix the lower the probability of accidental occurrences. When using a binary signalling medium, if the message prefix is k binary digits long, then the probability that a k-bit chunk, chosen at random from a random message, will accidentally equal the prefix is 2^{-k}. This relationship suggests that we can make the probability of accidental occurrences arbitrarily small just by making the prefix sufficiently long. This is only true if the message data is random, which may not always be the case. However, even in cases where we do have random message data, there is still a part of the transmitted data that is definitely not random at all — the message prefixes themselves.

Because we know that the transmitted data will contain many occurrences of the message prefix sequence, this sequence should be chosen carefully so that it does not interfere with its own detection. If the pattern is not chosen wisely then the message prefix can suffer from self-interference, which can make it impossible to tell where the exact frame boundary is supposed to lie. Consider the case where the message prefix is simply ten binary ones (a poor choice). The diagram below illustrates the tail end of one packet, followed by the message prefix and then the beginning of the next packet:

```
. . . 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 . . .
```

A receiver cannot tell what has been encoded here. The following interpretations are all equally correct:

```
. . . 0 0 0 [-Message--Prefix-] 1 1 0 0 0 . . .

. . . 0 0 0 1 [-Message--Prefix-] 1 0 0 0 . . .

. . . 0 0 0 1 1 [-Message--Prefix-] 0 0 0 . . .
```

The problem is that ten binary ones is a bit-pattern that has high auto-correlation: the pattern is very similar to itself shifted left or right a few bits. It is very easy for the tail portion of a message, concatenated with the start of the message prefix, (or the tail end of the message prefix, concatenated with the start of the next message) to look exactly like a message prefix. In this case, even though the probability of ten ones appearing somewhere in the middle of a random message is only $^1/_{1024}$, for mis-framing to happen right at the end of the previous message requires only that the last bit of the message is a one — a probability of $^1/_2$. Even if the message prefix were a hundred binary ones, the probability of mis-framing right at the end of the message would still be $^1/_2$. A long message prefix alone does not ensure that accidental occurrences will be rare. The message prefix must also have low auto-correlation.

2.3.5 Barker Codes and Comma-Free Encoding

In 1953 Barker studied the choice of in-band framing codes so as to minimize the possibility of self-interference, even in the presence of line noise [Bar53]. He made some specific recommendations of useful patterns, now known as Barker Codes, that have low auto-correlation, such as '110', '1110010', and '11100010010'. Using one of these patterns reduces the likelihood of mis-synchronization at the beginning or end of the message, but still doesn't guarantee that mis-synchronization cannot happen by accident somewhere within a message. Barker stated that he "assumes that the synchronizing signal will consist of a pattern of digits which is sufficiently unlikely to occur by chance in a random sequence or in a previous message." Hence Barker made an important contribution by showing that all framing patterns are not equally good, but he did not completely answer the question of how to provide reliable unambiguous framing for all messages, including unfortunate ones where the synchronizing signal does occur by chance.

At the time that Barker was doing his work on in-band framing codes, the contemporary literature [Gol58a] [Gol58b] [Kend62] [Gol63] [East65] [Sch69a] [Sch69b] referred to an encoding that can provide reliable unambiguous identification of message boundaries without resorting to an out-of-band framing marker as a 'comma-free' encoding, referring to the fact that the encoding needs no special reserved character (e.g. a comma) to separate multiple items/words/messages in a list. In a comma-free encoding the boundaries between the messages are implicit in the encoding, and no additional boundary markers are needed. In other words, no concatenation of symbols from the tail end of one message, joined to symbols from the beginning of another message, may appear to be itself an instance of a valid message.

Surprisingly, the first serious study of comma-free codes came not from the mathematical, communications, or computing communities, but from biologists attempting to gain an understanding of how deoxyribonucleic acid (DNA) encodes the necessary information for the synthesis of proteins. In 1958 Golomb, Welch & Delbrück [Gol58a] asked a most salient question, "If it is true that groups of base pairs, say triplets, code for one amino acid, and that these triplets are not overlapping, and if the message is formed simply by tacking these triplets end to end, how do we know where one triplet ends and where the next begins?" They go on to show how selection of an appropriate subset of 20 codes from the 64 possible triplets results in a comma-free encoding of amino acids. While this result was useful in the study of DNA, it only solved the problem for one particular set of short, fixed-size messages. It did not solve the general problem of how to communicate arbitrary variable sized messages.

A decade and a half after the biologists' work on comma-free codes, Artom expanded on Barker's work on synchronizing prefixes and showed how to algorithmically generate a comma-free encoding of any sequence of messages [Art72]. Artom showed that by adding 'stuffing bits' to the message data, any accidental occurrences of the prefix code could be eliminated. This stuffing process is reversible, so the receiver can correctly reconstruct the original message upon reception. This means that an appropriate non-self-interfering prefix, in conjunction with the use of stuffing bits, provides a programmatic way to generate a comma-free encoding. Recall that in a comma-free encoding no concatenation of symbols from the tail end of one message, joined to symbols from the beginning of another message, may appear to be itself an instance of a valid message. In Artom's scheme every message starts with exactly one instance of the prefix, and contains no other instances of the prefix. Hence any concatenation of symbols from the tail end of one message, joined to symbols from the beginning of another message, could not begin with the correct prefix and could not appear to be a valid message.

Artom also showed how to minimize the number of stuffing bits that may be required in the worst case. He expanded on Barker's constraints concerning the low auto-correlation of the prefix code, stating that both the prefix, and the prefix with its final bit removed, must be *valid sequences*. A sequence X is a *valid sequence* if X cannot occur in any overlap involving both X and a sequence of one or more random bits. For example, the sequence X=101 is not a *valid sequence,* because prepending 10 to it yields the sequence 10101, which contains an overlapping instance of X. The sequence X=110 is a *valid sequence* because there is no pattern of bits that can be prepended or appended to it to result in an overlapping instance of the same sequence. Artom shows that when both the prefix, and the prefix with its final bit removed, are *valid sequences*, the worst case for a prefix of length $k+1$ is one stuffing bit for every $k-1$ useful message bits.

To give a concrete example using Artom's terminology, HDLC (explained in more detail later in this chapter) uses the seven-bit message prefix 0111111. Both this prefix, and this prefix with the final bit removed, are *valid sequences* because appending bits after either 0111111 or 011111 results in an overlap region that does not begin with a zero, and consequently cannot be mistaken for the sequence. Similarly, prepending bits before either sequence results in an overlap region that contains a zero somewhere in the middle, and consequently cannot be mistaken for the sequence. Hence both this prefix and this prefix with the final bit removed are *valid sequences*, and since $k=6$, the worst case for HDLC is one stuffing bit for every five useful message bits.

Another decade and a half later, Papastavridis expanded on Artom's work by showing that the number of stuffing bits required on average is much less than the pessimistic worst case shown by Artom [Pap87]. Papastavridis shows that on average, for uniformly distributed random data, with a prefix of length $k+1$, the number of stuffing bits that has to be added is approximately one stuffing bit for every 2^k-2 useful message bits. Using the example of HDLC again, where $k=6$, this gives us an expected overhead of one stuffing bit for every 62 useful message bits.

In conclusion, it is only since the early 1970s that we have had communication protocols that provide transparent data transmission, and reliable resynchronization after error, without requiring an out-of-band framing marker.

2.4 Desirable Properties of a Framing Protocol

This dissertation identifies four desirable properties of framing protocols: Transparency, Reliability, Locality and Efficiency. The four properties are explained and justified below.

2.4.1 Transparency

A good framing protocol should be transparent to higher layers of software. They should not have any restrictions placed upon what data values they are allowed to send. Out-of-band framing protocols are necessarily transparent in that they use a framing signal that by definition can never appear in the data. For in-band framing protocols transparency is not automatic, so in-band framing protocols should use a data stuffing transformation to provide the necessary data transparency to higher layers of software.

Transparency is important because many data formats in common use, like JPEG [ISO10918] images, MPEG [ISO11172] video, and pkzip compressed files [Nel96], make full use of the entire range of 256 possible eight-bit byte values. To transmit these data types over a non-transparent medium would require the higher-layer software to be aware of the particular quirks of the medium, and to construct its data so as to comply with the restrictions imposed by the lower layer. Potentially, every application would need to be prepared to handle the quirks of every different non-transparent medium, so this is not a scaleable solution. This is an example of the $m \times n$ problem [Pad83]. Instead, the framing protocol should know its own quirks and should be responsible for seeing that any data sent is first converted into a format that does not violate the restrictions of the lower layer.

Some data types, such as plain ASCII text, may not use the entire range of available eight-bit byte values. In this case it might seem that some economy could be gained by sending the data directly over the non-transparent lower-layer without the expense of going through an unnecessary data stuffing step. This is a false economy because it applies only in the limited cases where the set of byte values that are not used in the data happens to coincide with the particular set of byte values that are prohibited by the lower layer. In other cases where the properties of the data do not match the details of the underlying medium's particular mode of non-transparency, data-conversion would still be required. In addition, any data type that never uses some of the available eight-bit byte values is not making efficient use of the storage medium, and an algorithm such as Huffman encoding [Huff52] [Knu85] which takes advantage of the non-uniformity in the distribution could be used to compress the data to a smaller size. After such compression, the resulting data does not, in general, leave any eight-bit byte values unused.

2.4.2 Reliability

This dissertation judges the reliability of a framing protocol by whether it is guaranteed to resynchronize after an error. All communications systems are subject to errors of some kind, and although a single error may cause the loss of some amount of data, it should not disable the communications system for an unbounded length of time. A similar issue is that a new receiver joining an existing data stream already in progress (such as a satellite receiver tuning to a new MPEG video channel) needs to be able to synchronize with the transmitter and determine where each new packet begins in the data stream.

Out-of-band protocols have no trouble resynchronizing after an error, because by definition their framing marker cannot appear anywhere in any packet. Any time the receiver detects a framing marker it serves as an unambiguous indication that a new packet is beginning. In contrast, many in-band framing protocols do not guarantee to resynchronize after an error.

Edsger Dijkstra showed that it is not an unreasonable requirement to expect a communications system to guarantee resynchronization. In his studies of *Self-Stabilizing Systems* [Dijk74] [Dijk86] he showed a very powerful result: a properly designed distributed system can be guaranteed to arrive at a correctly synchronized state in a finite length of time, even in a system with a large number of autonomous components and only sparse communication between those components, no matter what the initial state of the system's components. Given this powerful general result, it seems a fairly modest requirement to say that a single receiver should guarantee to synchronize with a single transmitter within some finite period of error-free communication. Nevertheless, many in-band framing protocols fail to achieve this simple capability.

Many in-band framing protocols offer only the weaker guarantee that they are likely to synchronize fairly quickly, most of the time, as long as the packet data does not contain too many unfortunate patterns that are likely to confuse the receiver. Although this kind of ‘probabilistic’ notion of reliability may be useful in some contexts, this dissertation adopts the stronger requirement that a framing protocol should always guarantee to resynchronize, no matter what data the packet contains.

2.4.3 Locality

A good framing protocol should make it straightforward for the receiver to detect where each packet begins in the data stream, both to simplify normal operation, and particularly to allow rapid recovery after an error. In this dissertation a framing protocol is said to have *locality* if the only requirement for a packet to be received correctly is that the entire packet and its immediately adjacent framing markers (the ones preceding and following it) are received without error. In other words all the information that is required to decode the packet data correctly is local to the packet — just the packet itself plus its immediately adjacent framing markers. For a framing protocol that has *locality,* no combination of insertion, deletion, or replacement errors preceeding the initial framing marker, nor following the final framing marker, can prevent the receiver from receiving and decoding the packet correctly.

Many framing protocols do not have locality. In many framing protocols, after an error the receiver has to examine a large amount of data before it is able to resynchronize and start decoding packets again. During this resynchronization period, a large number of packets may be received entirely correctly, only to be lost because of the receiver’s inability to recognize them. This lack of locality multiplies the effective error rate of the medium, because a single error results in several packets being lost instead of just one.

Another drawback of non-local framing is that it can make implementation more complicated. Because the information that encodes the synchronization signal is distributed throughout the transmitted stream instead of being localized, the receiver may need a complicated finite state machine or large memory buffers (or both) to recognize and decode this synchronization signal.

2.4.4 Efficiency

A good framing protocol should not add excessive overhead to the data being sent, because this makes inefficient use of the underlying medium. There are two components of overhead that a framing protocol adds. One is the overhead of the framing marker itself, and the other is the overhead of providing the necessary data-transparency.

A large framing marker (one that is many bits long) may yield a low probability of false detection in a bitstream that is subject to errors, but it also results in a high fixed overhead per packet. If most packets are large then this fixed overhead may be a small percentage of the medium's total bandwidth, but the smaller the average packet length, the larger the proportion of the bandwidth consumed by framing markers.

Out-of-band framing is by definition a transparent framing method, but as described by Robert Scholtz in [Sch80], this transparency is not without cost, and this effective cost increases with the packet length. The longer the packets, the longer the periods during which the out-of-band framing signal never appears, and during those periods the full signalling capability of the medium is not being utilized to its full extent.

With in-band framing, transparency is provided by data stuffing rather than by making incomplete use of the full signalling capabilities of the medium, but, as with out-of-band framing, the cost of that transparency tends to increase as packet length increases. For in-band framing the overhead also depends on the choice of framing marker, and tends to increase as the length of the framing marker decreases. This is because the shorter the framing marker, the more likely it is that that pattern will occur by chance in the data to be sent, resulting in stuffing events which add overhead [Art72] [Pap87]. This creates an interesting trade-off: For any given encoding algorithm, a short framing marker results in low framing overhead but high data stuffing overhead, and a longer framing marker results in higher framing overhead but reduces data stuffing overhead.

The properties of Transparency, Reliability and Locality are all binary properties. That is to say that a framing protocol either has the property, or it does not. In contrast the property of Efficiency is harder to quantify. As well as a fixed cost per packet, there may be a cost that varies depending on the size of the packet, and even depending on the contents of the packet. In addition, making comparison even harder, some framing protocols use only a single fixed frame size, some support variable-sized frames up to some maximum size, and some allow arbitrary-sized frames without limit. This dissertation quantifies the overhead using parameters α (average percentage overhead), β (worst-case percentage overhead) and k (constant overhead), as described below. The parameters α and β are expressed as a percentage of the unencoded packet size, and the constant overhead k is expressed in bits. How these parameters are calculated for fixed-size, finite-size, and arbitrary-size frames is explained below:

2.4.4.1 Fixed-Size Frames

For framing protocols that use a fixed size frame, this dissertation expresses the average amount of overhead as a percentage α of the amount of the user data carried in the frame, since this gives the most intuitive notion of how much of the underlying medium's capacity is being wasted. In cases where the amount of overhead is not fixed, and can vary depending on the contents of the frame (as is often the case with data stuffing algorithms) both the average percentage overhead α and the worst-case percentage β are given.

2.4.4.2 Finite-Size and Arbitrary-Size Frames

Some framing protocols use a variable size frame but have some intrinsic limit on the maximum frame length that can be encoded (e.g. because they use a fixed one-byte length field). Other framing protocols allow arbitrarily large frames. For example, although Ethernet [Tan88] [IEEE802.3] specifies a maximum packet size of 1514 bytes, the *framing protocol* used by Ethernet has no such intrinsic limit, and could be used to frame packets of any length we choose. For these kinds of framing protocol there is no obvious canonical packet size at which to evaluate the protocol's efficiency. While choosing the largest possible packet size might be fair in a theoretical sense, it would not be very useful, because for some protocols that would be a hypothetical infinitely large packet and in practice no network software or hardware uses infinitely large packets. It would also ignore the important differences between different protocols' fixed costs, because any fixed cost is reduced to zero when amortized over an arbitrarily large packet size. Similarly, trying to choose one particular 'typical' size of network packet is equally problematic, because network packet sizes vary widely, from 16 bits in Cambridge Ring (Section 2.9.6) to 4608 bits in Extended Superframe (Section 2.8.2), and even higher in protocols like Ethernet (Section 2.7.1) and Token Ring (Section 2.7.2). To avoid this dilemma, the overhead is divided into a fixed component which is independent of packet size, and a variable component which varies linearly with packet size. The fixed component of overhead is expressed as a number of bits k, and the variable component is expressed as above, as an average percentage α and (when different) a worst-case percentage β.

2.5 The Framing Hierarchy

Usually a communications system will have several levels of message framing, for example the physical hardware level converting a continuous analogue signal into discrete binary bits, the bits-to-bytes level, and the bytes-to-packets level. Some systems combine all the levels, going directly from physical signals to packets in a single step, and others are more intricate, having even more than three layers of framing.

At the lowest level we often find out-of-band protocols. They use specific physical properties, like voltages or timing information, that are outside the control of higher layers.

Moving higher up the protocol stack, other framing protocols operate in terms of binary digits rather than directly in terms of physical properties. These framing protocols trust a lower layer of the stack to convert physical measurements of the medium into abstract digital values, and then operate using only those digital values. Since the alphabet these protocols use, $\{0,1\}$, is the same alphabet that all their software clients at the higher layers also use, they are in-band framing protocols.

Moving still higher up the protocol stack, other framing protocols operate in terms of units that are eight-bit bytes. These framing protocols trust a lower layer of the stack to decode and present those eight-bit byte values correctly. In principle a single physical measurement of the medium could yield an entire byte value, but it is far more common for each measurement of the medium to yield a smaller quantity of information such as a single one-bit binary value. When the underlying physical layer is delivering a stream of bits, and the framing protocol operates in terms of a stream of bytes, there is an implicit assumption that there is also some intermediate layer that is responsible for correctly determining where the boundaries between bytes should fall in the unstructured bit-stream. This intermediate layer is itself a framing protocol (bits-to-bytes), and the correct operation of all the higher layers depends on its correct operation. In many cases, the layer that performs the bits-to-bytes framing is an RS-232 serial port. As explained in

Section 2.9.3, RS-232 framing is not guaranteed to recover after an error, although in practice it will eventually resynchronize after enough bytes have passed.

Moving even higher, protocols like TCP could be considered to be elaborate framing protocols, taking multiple IP packets and grouping them into units called ‘TCP connections’, but such analysis goes beyond the scope of this dissertation.

2.6 Review of Existing Framing Protocols

The remainder of this chapter considers in more detail the practical framing and encoding schemes that have been used in the past, or are in use today, for actual computer communications systems. Because there are so many of them, it is useful to organize them into four families: out-of-band framing, non-transparent in-band framing, transparent in-band *without* data stuffing, and transparent in-band framing *with* data stuffing. The relationships between the four families are illustrated in Figure 1.

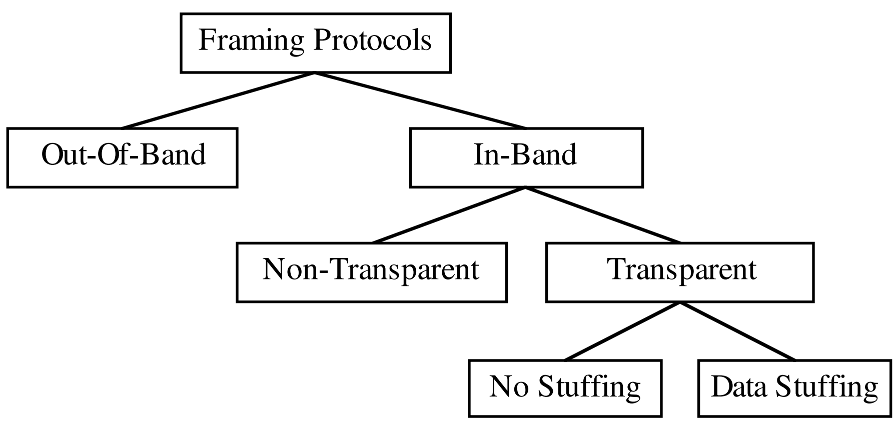


Figure 1. Taxonomy of Framing Protocols

The protocols are described in the following approximate order (the term ISDN actually encompasses several different protocols, which for simplicity are all described in the text under a single ‘ISDN’ heading):

1. Out-of-band framing (Ethernet, Token Ring, FDDI, Teletext, ISDN-S)
2. Non-transparent in-band framing (MPEG, T-1/AMI)

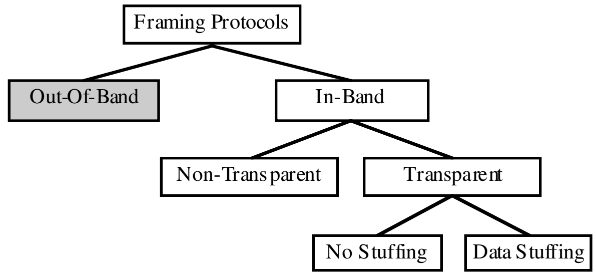
3. Transparent in-band framing without data stuffing (T-1/B8ZS, T-1/Verilink, ISDN-U, Baudot, RS232, SONET, Cambridge Ring, RATP, DDCMP, XMODEM, YMODEM)
4. Transparent in-band framing with data stuffing (IBM BISYNC, Kermit, ZMODEM, SLIP, AX.25, PPP, Gill's Protocol, HDLC).

For each of the different protocols I present a table summarizing:

1. Whether it uses in-band framing
2. Whether it offers data transparency
3. Whether it reliably resynchronizes after an error
4. Whether it has locality of framing information
5. The underlying message unit which is being framed into higher level messages
6. The size or size range of the higher level messages
7. How much overhead the framing protocol adds to the message

Items 1-4 are desirable properties of a framing protocol. For item 5, protocols that can frame individual bits or physical signals into packet are more versatile than ones that work on bytes. Protocols that work on bytes either need to operate over an inherently parallel medium at least one byte wide (e.g. a SCSI interface [X3.131]), or rely on having some other framing protocol to frame individual bits or physical signals into bytes for them, before they can frame those bytes into packets. Still, most framing protocols work on bytes, so we cannot ignore byte-framing protocols. For item 6, framing protocols that can be used to frame messages of any length are desirable because they have wider applicability than protocols that have some inherent message size limit. For protocols that meet all requirements of a particular application, item 7 is the tie-breaker that measures how efficiently they perform.

2.7 Out-Of-Band Framing Protocols



This section describes Ethernet, Token Ring, FDDI, Teletext, and ISDN.

2.7.1 Ethernet

Ethernet uses Manchester Encoding to encode the packet data [Sta94] [Tan88] [IEEE802.3]. In Manchester Encoding, each bit in the packet is effectively encoded as two bits on the wire. A binary one in the packet is encoded as a 01 on the wire, and a binary zero in the data is encoded as a 10.

Manchester Encoding has a potential framing ambiguity. If the receiver observes the signal 010101010101010 on the wire, how can it tell if it is observing …01, 01, 01, 01, 01, 01, 01, 0… (a sequence of encoded ones) or …0, 10, 10, 10, 10, 10, 10, 10… (a sequence of encoded zeroes)?

A careful choice of transmitted data can help eliminate ambiguity in the received signal. Ethernet takes advantage of this and begins each network packet with a preamble consisting of the byte value 10101010 repeated seven times and then the value 10101011. The value 10101010 encodes as 0110011001100110. Since we know that 00 and 11 are not legal Manchester Encoded values, the interpretation …0, 11, 00, 11, 00, 11, 00, 11, 0… is clearly invalid, so the receiver knows that the correct interpretation can only be 01, 10, 01, 10, 01, 10, 01, 10, and this allows it to synchronize correctly.

Ethernet has no provision to prevent the preamble sequence from occurring within the body of a packet. Ethernet solves this problem by enforcing a pause between packets. The Ethernet framing marker is therefore not defined by just a preamble sequence, but by a pause followed by a preamble sequence. Because every bit in the body of a packet always encodes as a bit pair containing a transition (either 01 or 10) there is no way for any data in the body of a packet to encode as a sequence with no transitions and masquerade as a pause.

In-Band Framing	✗
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Physical Signals
Message Size	Unlimited
Overhead	α = 100%* k = 64 bits

Table 1. Summary of Framing Properties for Ethernet

* Since Manchester Encoding transmits two voltage levels for each single bit, it is effectively using only half the raw data rate of the medium. The benefit of accepting this low efficiency is simplicity in implementation. Clock recovery, for instance, is easy in Manchester Encoding.

2.7.2 Token Ring

Token Ring [IEEE802.5] uses Differential Manchester Encoding to encode the packet data [Sta94] [Tan88]. Differential Manchester Encoding is similar to standard Manchester Encoding except that a zero-bit is indicated by repeating the previous two-bit pair (either 01 or 10), and a one-bit is indicated by outputting the opposite of the previous two-bit pair. This change of encoding rules gives the transmitted signal slightly different properties which are thought to be beneficial in some circumstances [Sta94], but just as with straight Manchester Encoding, every bit in the body of a packet always encodes as a bit pair containing a transition (either 01 or 10).

Rather than use a pause and a preamble like Ethernet, Token Ring uses a protocol violation — an out-of-band signal — to mark the start of the packet. Each packet is preceded by a special sequence which is the same length as a normal eight-bit byte, but contains pairs 11 and 00 which are normally invalid and can never appear in the encoding of the packet data itself.

In-Band Framing	✗
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Physical Signals
Message Size	Unlimited
Overhead	$\alpha = 100\%$* $k = 8$ bits

Table 2. Summary of Framing Properties for Token Ring

* Since Manchester Encoding transmits two voltage levels for each single bit, it is effectively using only half the raw data rate of the medium. The benefit of accepting this low efficiency is simplicity in implementation. Clock recovery, for instance, is easy in Manchester Encoding.

2.7.3 Fiber Distributed Data Interface

Fiber Distributed Data Interface (FDDI) is in many ways similar to Token Ring (Section 2.7.2) at the higher layers, but at the lower layers it is quite different. One obvious difference is that the data rate is higher, but more significantly the framing and encoding are also different.

FDDI uses a 4b/5b block encoding, which means that each four bits of data are encoded as a five-bit sequence transmitted over the medium [Shah93]. This 4b/5b encoding gives FDDI an interesting two-tier framing problem to solve: The incoming bit-stream has to be correctly framed into five-bit symbols, and the resulting stream of five-bit symbols has to correctly framed into larger FDDI packets. Rather than solving these two framing problems independently, which would be less efficient, FDDI uses an encoding that allows it to solve both in a single step.

Like T-1 (Section 2.8.2), FDDI needs a certain number of transitions in the data in order to maintain clock synchronization. In the 4b/5b encoding, there are 32 different five-bit symbols available, and since only 16 different four-bit values need to be represented, this allows FDDI to be somewhat selective about which of the 32 possible symbols it uses. To maintain a high rate of transitions FDDI encodes data values using only symbols that have two, three, or four one-bits, and no more than two consecutive zero-bits. Other symbols are either declared invalid and not used, or are used for special control functions, such as error signalling and packet framing. Table 3 shows the meanings assigned to each of the 32 symbols.

Framing in FDDI is performed using the following sequence:

Preamble (16 Idle symbols), Start Delimiter (JK symbol), Frame Control (1 byte), Destination Address (6 bytes), Source Address (6 bytes), Payload (up to 4478 bytes), Frame Check Sequence (4 bytes), End Delimiter (T symbol), Frame Status (3 symbols).

FDDI data is transmitted over the physical medium using NRZI (non return-to-zero inverting) encoding. NRZ means that the bits are transmitted continuously without gaps between them — there is no pause between bits where the signal returns to zero. NRZI is a modified form of NRZ where a one-bit is transmitted by inverting the output, and a zero-bit is transmitted by leaving the output unchanged [Sta94].

Symbol	Meaning	Symbol	Meaning
00000	Code: Quiet	10000	Code: Unused/Halt
00001	Code: Unused/Halt	10001	Code: K
00010	Code: Unused/Halt	10010	Data: 0x8
00011	Code: Unused	10011	Data: 0x9
00100	Code: Halt	10100	Data: 0x2
00101	Code: Unused	10101	Data: 0x3
00110	Code: Unused	10110	Data: 0xA
00111	Code: R (Reset)	10111	Data: 0xB
01000	Code: Unused/Halt	11000	Code: J
01001	Data: 0x1	11001	Code: S (Set)
01010	Data: 0x4	11010	Data: 0xC
01011	Data: 0x5	11011	Data: 0xD
01100	Code: Unused	11100	Data: 0xE
01101	Code: T	11101	Data: 0xF
01110	Data: 0x6	11110	Data: 0x0
01111	Data: 0x7	11111	Code: Idle

Table 3. 4b/5b Codes for FDDI

The FDDI preamble consists of 16 Idle symbols, which results in a long unbroken burst of ones to be transmitted, which causes the output to be inverted for every bit. This high-frequency stream of on-off pulses enables the receiver to reliably synchronise its bit-level clock to the transmitter, and also serves as an out-of-band signal to reset the state of the receiver in preparation for receiving a new packet. This preamble is followed by the start delimiter, which is symbol J (11000) followed by symbol K (10001), which gives the binary sequence 1100010001. Not only is the start delimiter preceded by a long stream of binary ones, which can never occur inadvertently in a stream of data symbols, the JK sequence itself is also a pattern that can never appear inadvertently in a stream of data symbols. For a mis-framed series of data symbols to masquerade as the start delimiter, it would have to contain one of the following four sequences:

```
x1100 01000 1xxxx

xx110 00100 01xxx

xxx11 00010 001xx

xxxx1 10001 0001x
```

The first three sequences can never occur in valid data, because they contain invalid (unused) symbols, and the fourth contains the K symbol, which also can never occur in the body of a valid data packet. Hence, any time the receiver detects the start delimiter sequence 1100010001, it can discard any packet partially received up to that point and enter a state in which it is ready to receive the next complete FDDI frame. The JK start delimiter simultaneously provides bit-level framing to identify symbol boundaries in the data stream, and symbol-level framing to identify where a new FDDI frame begins in the symbol stream. At the bit-level this JK pair can be thought of as an in-band framing signal, applied to a payload that has had a reversible data transformation applied to it to eliminate any inadvertent occurrences of the in-band framing signal. Equivalently, at the symbol level, FDDI framing can be thought of as an out-of-band framing signal, using a special reserved symbol which is not available to the higher layers.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Physical Signals
Message Size	Unlimited
Overhead	$\alpha = 25\%$ $k = 280$ bits

Table 4. Summary of Framing Properties for FDDI

2.7.4 Teletext

Teletext [Vei83] is a transmission mechanism that encodes textual information in unused scan lines of television signals. Data is transmitted in fixed size packets which usually contain a single line of text to be displayed on the viewer's screen. There are a number of variants of Teletext in use around the world, but a typical Teletext packet is formed as follows: It begins with the television's horizontal scan line synchronization pulse, followed by the analogue 'colour burst' information (used for colour television pictures), the bit synchronization pattern (16 bits of alternating 1s and 0s), the byte synchronization pattern (e.g. 11100111), a five-byte prefix (header) and then 40 bytes of data (the text). The transmission is synchronous — there are no start/stop bits around each byte as there are with RS-232 (see below), nor is there any need for any bit- or byte stuffing. The line synchronization pulse, which always precedes the beginning of a new Teletext data packet, is a negative pulse that goes below the normal 'black' level that may appear anywhere within the image portion of any scan line. Thus Teletext is using an out-of-band signal to mark packet boundaries.

In-Band Framing	✗
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Physical Signals
Message Size	Fixed (one line of text)
Overhead	$\alpha = 20\%$

Table 5. Summary of Framing Properties for Teletext

2.7.5 ISDN

Although ISDN is often thought of as a single international standard, the term ISDN actually encompasses a whole suite of protocol standards [Helg91]. In its full form, an ISDN connection consists of a Local Exchange (LE) connected via local loop to a Network Terminator-1 (NT1) device in the customer premises, connected to a Network Terminator-2 (NT2) device, connected to a Terminal Adapter (TA) connected to the customer's Terminal Equipment (TE). The interface between TE and TA is called the R reference point; the interface between TA and NT2 is called the S reference point; the interface between NT2 and NT1 is called the T reference point; and the interface between NT1 and LE is called the U reference point. This architecture is shown in Figure 2.

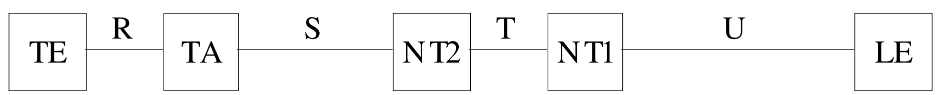


Figure 2. ISDN Terminology and Reference Points

2.7.5.1 ISDN 'R' Reference Point

The ITU standards do not specify the R reference point, other than noting that it should use the protocol appropriate for communicating with the attached Terminal Equipment, usually RS-232. These days, when the attached device is more likely to be a computer than a teletypewriter or dumb terminal, 10Base-T Ethernet is a better choice for the R reference point protocol.

2.7.5.2 ISDN 'S' Reference Point

The S reference point defines a local communications protocol within the customer premises. The S reference point uses pseudoternary encoding, also known as bipolar or modified alternate mark inversion encoding [Bea90]. Pseudoternary encoding is a bipolar (three-state) encoding protocol where one-bits are encoded as zero volts on the line, and zero-bits are encoded alternately as positive and negative pulses. This alternating voltage encoding maintains the line's DC balance (no net DC current) which gives it good signal properties, such as the ability to pass through transformer couplings. Long strings of zero-bits encode as alternating positive and negative pulses which give regular voltage transitions for the receiver to synchronize on, but long strings of one-bits result in no activity on the wire, which can cause the receiver to drift and lose bit-synchronization. The S reference point avoids this problem by placing framing bits and other control information after every eight-bits of user data, thereby limiting the time that can elapse without any transitions on the wire. The framing is performed using a bipolar violation — two consecutive voltage pulses with the same sign. Since this pattern of signals is illegal for normal data, this framing signal cannot occur inadvertently in the data stream, so this is an out-of-band framing mechanism.

Each frame consists of 48 bits, containing two eight-bit samples each from the B1 and B2 data channels, four bits of data from the D channel, and twelve other overhead bits, including framing bits. Since in practice the 16kb/sec D channel is not used, this gives us 32 bits of user-data and 16 bits of overhead per frame.

In-Band Framing	✗
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Physical Signals
Message Size	Fixed (32 bits; four samples of 8 bits each)
Overhead	α = 50%

Table 6. Summary of Framing Properties for ISDN 'S' Reference Point

2.7.5.3 ISDN 'T' Reference Point

In most cases the NT1 and NT2 are combined into a single device, so the T reference point has become redundant. When it does exist as a separate interface, it uses the same framing structure as the 'S' Reference Point [Sta95].

2.7.5.4 ISDN 'U' Reference Point

The ITU standards also do not specify details of the U reference point [Kes96]. The assumption in Europe was that the NT1 would be owned and operated by the telephone company, so the details of how it interfaced with the LE would be of concern only to that telephone company, and not subject to international standardization. The United States government disagreed, wishing to promote competition and allow the subscriber the option of owning the NT1 if so desired [T1.601].

In the United States of America, ANSI defined a standard for the U reference point [T1.601]. The signalling scheme is called two binary, one quaternary (2B1Q). 2B1Q is a four-level line code, with each quaternary symbol (quat) carrying two bits of information. Framing is performed using a nine quat 'synchronization word' which appears at the beginning of each frame. Representing the four quat values by –3, –1, +1, +3, the synchronization word is the sequence +3 +3 –3 –3 –3 +3 –3 +3 +3. This synchronization word is an in-band framing signal. No measures are taken to ensure that this sequence does not inadvertently appear in the data payload, making this framing error prone, and potentially vulnerable to malicious attack. Also, this encoding is not guaranteed to have good DC balance. The ITU standards specify a scrambling procedure to be applied to the data in an attempt to circumvent these problems. While this makes pathological packets harder to construct deliberately, it does not fundamentally alter the fact that such pathological packets do still exist.

Following the nine quat (18 bit) synchronization word, there are 216 data bits containing 96 bits from each B channel and 24 bits of D channel information, followed by 6 bits of the M (network maintenance) channel, making a total of 240 bits per frame. Since in practice the 16kb/sec D channel is not used, this gives us 192 bits of user-data and 48 bits of overhead per frame, a 25% overhead.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Quats (2 bits per symbol)
Message Size	Fixed (192 bits)
Overhead	$\alpha = 25\%$

Table 7. Summary of Framing Properties for U.S. ISDN 'U' Reference Point

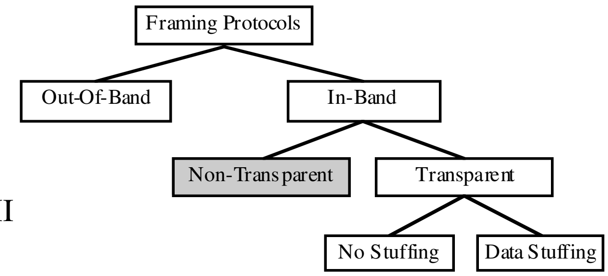
Outside the United States of America, a ternary encoding for the U reference point is common [Kahl86]. Using MMS43 (Modified Monitoring Sum / Modified Monitoring State four binary three ternary) groups of four binary digits are encoded using three ternary symbols. Because three ternary symbols encode 27 different values, and only 16 are needed to encode four binary digits, this encoding has some redundancy. The encoding is designed, taking advantage of that redundancy, to avoid sequences that have poor DC balance. The eleven element Barker code + + + – – – + – – + – is used as the synchronization word, and just as in the ANSI standard, no measures are taken to ensure that this sequence does not inadvertently appear in the data payload. One might imagine that, given the redundancy of the MMS43 encoding, the synchronization word would be chosen to be one that could never appear as a result of encoding binary data, but this is not the case. For example, the binary sequence 1100 1010 1100 1100 encodes as + + + + – – – + – – + –, which inadvertently contains the synchronization word sequence.

Following the eleven element synchronization word, there are 216 data bits. The 216 data bits are encoded using 162 ternary symbols, making a total of 173 ternary symbols. 216 data bits could be encoded using just 137 ternary symbols, meaning that this encoding has an overhead of about 26.3%. However, this analysis ignores the factor that MMS43 encoding helps ensure DC balance, which is important. If we consider just the eleven element synchronization word as overhead on 173 data symbols, this gives us an overhead of about 6.8%.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bits
Message Size	Fixed (192 bits)
Overhead	$\alpha = 6.8\%$

Table 8. Summary of Framing Properties for Non-U.S. ISDN ‘U’ Reference Point

2.8 Non-Transparent In-Band Framing Protocols



This section describes MPEG and T-1/AMI Framing.

2.8.1 MPEG

MPEG [ISO11172] is usually thought of as a video compression algorithm, but MPEG was also designed to be used as a raw network protocol, even carrying other data in addition to audio and video frames.

MPEG frames its blocks using a start-of-block marker, and also has a length value in the block header to indicate how long the block is. The start-of-block marker is three zero bytes followed by a one-byte 'block designator' which indicates what kind of block it is (video frame, audio frame, data frame, etc.).

MPEG does not perform any kind of data stuffing to eliminate occurrences of zero triplets from the data. Instead, the MPEG committee defined the MPEG video encoding very carefully so that under no circumstances can three zero bytes ever appear anywhere in the body of an encoded video frame.

The issue of new receivers joining at arbitrary times is of particular importance to MPEG. With satellite MPEG video broadcasts, new receivers join at arbitrary times and need to be able to recognize where the next MPEG block begins in the transmission. They can achieve this synchronization by observing the video data stream waiting for the next appearance of three zeroes. Similarly, video CDs are simply a flat-file format with no indexing to indicate where each MPEG data block begins on the disk. To perform a random access seek to some place within the MPEG file, it is necessary first to seek to some arbitrary location within the file and then to search linearly forward to find the three zeroes that indicate the beginning of the next video block [May96].

Unfortunately, while the MPEG committee defined the MPEG video encoding so that three zero bytes never appear in the body of a video frame, they did not do the same for audio blocks. In practice, it turns out that spurious occurrences of three zero bytes occur surprisingly often in audio blocks [May96]. In audio-only MPEG streams the problem is especially acute. Simply detecting a sequence of three zero bytes does not give sufficiently high confidence for the MPEG player to begin decoding immediately. To synchronize reliably, the MPEG player must not only find a sequence of three zero bytes, but it must also read the length from the block header, read that entire block, and then examine the following block as well to see if that one also looks valid. Only then does it have sufficiently high confidence that it has really decoded the MPEG data correctly. This reading ahead requires extra memory buffers (MPEG blocks can be up to 64kbytes each) and takes extra time. In demanding applications it may be necessary to read ahead two or three or even more blocks to get the required confidence, and even then there is still no guarantee that the data is decoded correctly. There may be a whole stream of audio blocks where an accidental run of zeros occurs in an unfortunate location within each block.

In-Band Framing	✔
Transparency	✘ *
Reliability	✘ *
Locality	✘ *
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 0\%$ $k = 32$ bits

Table 9. Summary of Framing Properties for MPEG

* If MPEG data is not allowed to contain runs of three zeroes, then MPEG framing is not transparent. If the data does happen to contain runs of three zeroes, as audio blocks do, then MPEG framing is not reliable. If greater reliability is to be achieved by making the framing algorithm look many blocks into the future to get a reasonably high statistical confidence of proper framing, then MPEG framing lacks locality.

2.8.2 T-1 / D4 Framing with AMI Line Coding

Like 'ISDN', the term T-1, as it is commonly used, actually encompasses a whole suite of standards. Strictly speaking, the term T-1 means just an unstructured full-duplex bit-stream at a rate of 1.544Mb/sec running over two twisted pairs of copper wire. The interface that is presented to the user of the T-1 service is called DS-1 ('Digital Signal, level 1') [Sta95]. The DS-1 service provides the user with 24 separate 'channels' of data, which, in conventional use, correspond to 24 separate telephone calls. The DS-1 service aggregates data from the 24 channels into fixed size frames which are sent over the T-1 line. Each frame consists of a single framing bit, and eight bits of data from each of the 24 channels, making a total of 193 bits. In some cases all eight bits of data per channel are available for user data, giving a total usable transmission rate of 1.536 Mb/sec. In older systems, only seven bits of data per channel are available for user data, giving a usable transmission rate of only 1.344 Mb/sec. Two common ways of providing the DS-1 service over a T-1 line are 'D4' framing with AMI (Alternate Mark Inversion) line coding, and ESF (Extended Superframe) framing with B8ZS (Bipolar with 8-Zeros Substitution) line coding [Mauf97].

In the simplest DS-1 framing, the framing bit is alternately a one-bit and then a zero-bit in each successive frame [Mil87]. This alternating pattern is uniquely recognizable in the telephone system, because for any of the voice channels in the DS-1 frame to generate a pattern of alternating ones and zeroes in successive samples, it would have to be carrying an audio tone of exactly 4kHz, and that is not possible because the analogue telephone system filters out all tones of 3.4kHz or higher [Kes96,p18]. In the analogue context this alternating zero/one pattern is an out-of-band framing signal. With the increasing use of computer communications, this assumption is no longer safe. A computer can easily generate any sequence of binary digits it desires, so in the digital context the T-1 framing protocol is an in-band framing signal that could quite easily be mimicked by the traffic being carried.

In D4 framing, 12 basic frames are grouped into a 'superframe' [Held91]. Rather than simply alternating between ones and zeros, the framing pattern for a 12-frame superframe is the sequence 100010011101 (sometimes described as being the sequence 101010 and the sequence 000111 interleaved). Just as with basic DS-1 framing, there is no protection against this framing pattern inadvertently appearing within the data of the frame.

Alternate Mark Inversion encoding is a bipolar (three-state) line code where zero-bits are encoded as zero volts on the line, and one-bits are encoded alternately as positive and negative pulses. This alternating voltage encoding maintains the line's DC balance (no net DC current) which gives it good signal properties, such as the ability to pass through transformer couplings. Long strings of one-bits encode as alternating positive and negative pulses which give regular voltage transitions for the receiver to synchronize on, but long strings of zero-bits result in no activity on the wire, which can cause the receiver to drift and lose bit-synchronization. For this reason, two rules must be followed when sending data over a T-1 line using D4/AMI: there must never be more than 15 zero-bits in a row, and in any sequence of $8n$ data bits there must be at least n–1 one-bits. For this condition to be satisfied it is sufficient to enforce a simpler channel-level constraint: no channel may contain the value zero. If no channel contains the value zero, then every channel must contain a value that has at least one bit set. If every channel contains at least one one-bit, then it can contain at most seven zero-bits, and there can never be a run of more than 14 zero-bits in a row, and the overall number of one-bits in the data will also be sufficient. When sending audio data (telephone calls) this channel-level constraint is easily met by simply setting the least-significant bit of any channel that contains the value zero (i.e. changing the byte value from 0 to 1). This conversion of channels with the value 0 to the value 1 imperceptibly degrades the audio quality, but ensures that the T-1 transmission rules are not violated. For computer data, this simplistic corruption of the transmitted data is not acceptable, so it is common when using D4/AMI that only seven of the eight bits in each channel are available for user data; the network interface hardware (CSU) is able to set the least-significant bit of each channel to one, if necessary, to ensure that the above condition is satisfied [Held91].

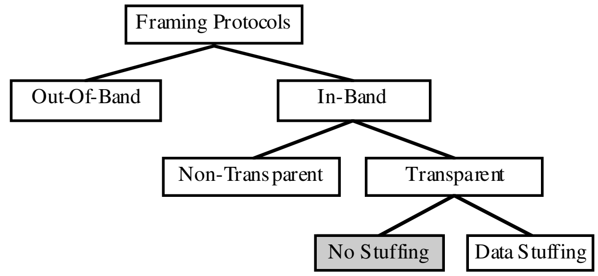
In-Band Framing	✓
Transparency	✗ (data must not violate AMI rules)*
Reliability	✗
Locality	✗
Underlying Unit	Bits
Message Size	Fixed (192 bits)
Overhead	$\alpha = 0.52\%$

Table 10. Summary of Framing Properties for D4 Framing with AMI Line Coding

* When adopting the convention that only seven of the eight bits per byte may contain user data, D4 Framing with AMI Line Coding gains data transparency, but the overhead increases from 0.52% to 14.88%.

It is also arguable that there is some additional degree of inefficiency in the fact that the line is capable of signalling tri-state information, but each tri-state signal is only being used to communicate a bi-state (binary) symbol of information. The exact extent of this inefficiency is debatable, because the need for DC balance and clock recovery means that every symbol cannot be chosen without constraint from the ternary alphabet in order to communicate $\log_2(3)$ bits of information per symbol. However, the argument that on most lines there is some additional signalling capacity untapped by D4/AMI is supported by the fact that on the same lines ESF Framing with B8ZS Line Coding (below) is able to provide transparent communication without having to discard one bit in eight.

2.9 Transparent In-Band Framing Protocols Without Data Stuffing



This section describes T-1/B8ZS, T-1/Verilink, Baudot Code, RS232, SONET, Cambridge Ring, RATP, DDCMP, XMODEM and YMODEM.

Because these protocols perform no data stuffing, they have no component of overhead that varies with packet size, and no component of overhead that varies depending on packet contents. However, they all add some header information to every data packet, so they all do have overhead. For the protocols that use fixed-size packets, this overhead is expressed as the percentage α of the packet size, and for the protocols that allow variable-size packets, this overhead is expressed as a constant overhead *k*.

2.9.1 T-1 / ESF Framing with B8ZS Line Coding

The T-1 non-transparent AMI framing mechanism described above has largely been supplanted by a newer transparent framing mechanism called ESF/B8ZS. An extended superframe (ESF) is 24 consecutive 193-bit DS-1 frames [Held91]. The sequence of framing bits is:

```
d c d 0 d c d 0 d c d 1 d c d 0 d c d 1 d c d 1
```

The ‘d’ bits are used by the telephone company to perform network monitoring, set alarm conditions, and perform other control operations. The ‘c’ bits contain a cyclic redundancy check sequence. Only 6 bits out of 24 have fixed values in order to facilitate synchronization of the receiver with the sender. As with D4, no precautions are taken to guard against this pattern appearing in the data stream. Furthermore, since some of the framing bits are now used to carry network management control information, if the user does construct a pathological pattern that inhibits re-synchronization after an error, the user is also able to deliberately send data that masquerades as ‘d’ bits, thereby taking over control of that part the telephone system’s switching fabric.

B8ZS is a bipolar line code like AMI, but with one significant difference. Like AMI, zero-bits are encoded as zero volts on the line, and one-bits are encoded alternately as positive and negative pulses. The difference is that in AMI long runs of zeroes cause no signal on the line, potentially causing the receiver to lose bit-synchronization. B8ZS avoids this problem by having an explicit encoding for a run of eight zeroes. B8ZS encodes a run of eight zeroes using the sequence:

0 0 0 + – 0 – + (if the last one-bit was sent as a + signal), or

0 0 0 – + 0 + – (if the last one-bit was sent as a – signal).

Because the voltage pulses are normally supposed to alternate between positive and negative, the receiver detects this double protocol violation and recognizes it as the special ‘eight zeroes’ code [Sta95].

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✗
Underlying Unit	Bits
Message Size	Fixed (192 bits)
Overhead	$\alpha = 0.52\%$

Table 11. Summary of Framing Properties for ESF with B8ZS Line Coding

2.9.2 Verilink Patent

In 1986, Marc L. Schas and Steven C. Taylor patented a technique for transmitting arbitrary binary data over a D4/AMI T-1 telephone circuit [Ver86].

As described in Section 2.8.2, two rules must be followed when sending data over a T-1 line using D4/AMI: there must never be more than 15 zero-bits in a row, and in any sequence of $8n$ data bits there must be at least n–1 one-bits. The mechanism in the Verilink patent provides a way of performing a non-destructive transformation on the data to ensure that these conditions are met.

A T-1 frame carries 24 channels of data, one eight-bit byte per channel. In each frame, the Verilink algorithm examines the 24 channels to see if any contain zero bytes. If none does, then the T-1 frame is transmitted as normal. If any do contain zero bytes, then the frame is transformed as follows:

The first thing that is done is that the single T-1 framing bit is inverted to indicate that this is a special frame. Then the channel numbers of the N channels that contained zero bytes are stored in the first N channels of the outgoing T-1 frame. The most significant bit of each byte in the first N–1 channels is set to one, each indicating that the following channel contains another similar channel number of a zero-containing channel, rather than an actual data value. The most significant bit of the byte in the last of the N channels is not set, indicating that it is the last of such channel-number channels in this frame. The following 24–N channels are filled with the values from the 24–N channels that contained non-zero bytes. This encoding allows the receiver to reverse the process and reconstruct the original frame.

This algorithm has two drawbacks. The first is that it is only applicable to communications systems using frames of 127 bytes or fewer. This constraint is not a problem for T-1 frames since they are never any size other than exactly 24 bytes long, but it limits the Verilink algorithm's more general applicability to other protocols. The second, more serious drawback, is that it corrupts the values of the T-1 frame's framing bits. When the receiver is perfectly synchronized, it can interpret a framing error indication as an indication of a specially encoded frame, but if the framing bit is now serving this alternative purpose, no mechanism remains by which the receiver can synchronize in the first place.

In-Band Framing	No framing at all *
Transparency	✓
Reliability	✗
Locality	✗
Underlying Unit	Bytes
Message Size	Variable, up to 127 bytes
Overhead	$\alpha = 0\%$ $k = 1$ bits

Table 12. Summary of Framing Properties for Verilink Framing

* Verilink Framing has no framing at all. It takes over the DS-1's framing bits for its own use, leaving the receiver no way to resynchronize after an error.

2.9.3 Baudot Code

Some 36 years after the invention of Morse Code in 1838, Jean-Maurice-Émile Baudot developed a new telegraphic code called the Baudot Code in 1874 [Brit56-97]. The two key features of Baudot's method were that it used mechanical time division multiplexing, and, unlike Morse Code, every symbol was represented using a fixed-length five-bit binary code [Cro08].

Baudot's equipment used a ring of metallic segments and a rapidly rotating armature which made brief contact with each segment in turn. By setting up an appropriately synchronized rotating armature at the receiving end, a voltage applied to any particular metallic segment of the ring would be conducted only to the circuit connected to the corresponding metallic segment of the receiving ring. By spinning the armatures rapidly enough, and using capacitors and inductors to smooth out the rapid pulsation of the signal, a single physical line was made to behave like multiple separate lines. With the spinning armatures in proper synchronization, multiple Morse Code operators could use these logical lines simultaneously without interfering with each other.

To keep the receiving armature in correct synchronization with the sending armature, 'Correcting Currents' were used. One special segment of the sending ring always had a voltage applied to it. When the receiving ring was in perfect synchronization, this Correcting Current would be delivered to a segment on the receiving ring that was connected only to ground, and the Correcting Current would have no effect. The segment immediately following this grounded segment was connected to a magnetic brake which would slightly slow the rotation of the armature. The rate of the receiving armature was set such that, in the absence of any correcting influence, it would run slightly too fast. Thus it would have a tendency to run ahead of the sending armature, and whenever it did so, a brief current would be applied to the braking segment, slowing the armature down again to keep it in perfect synchronization with the sending armature.

Baudot went one step further than simply using his device to allow multiple Morse Code operators to share a single line, by ganging together five logical lines into a single five-bit wide parallel communication channel, over which five-bit binary codes were used to represent letters of the alphabet, numbers and punctuation. There were actually two five-bit alphabets, one containing letters, and one containing digits and punctuation. In both alphabets, the code 00010 indicated that the following values should be interpreted as ‘figure codes’ and the code 00001 indicated that the following values should be interpreted as ‘letter codes’. In neither alphabet was the value 00000 used, since this was the value that was transmitted when the device was simply idle, with no keys depressed.

While in the simple Morse Multiplexer the slow rate of Morse Code keying was decoupled from the much faster rate of rotation of the armature, this was not the case for Baudot Code signalling. In Baudot Code signalling one symbol was transmitted for each rotation of the armature. If the keys were held down for two rotations of the armature, two identical characters would be printed at the receiving end. Thus the Baudot apparatus was essentially using the line in a synchronous bitwise serial fashion. The synchronizaton was maintained by the ‘Correcting Currents’, but since these Correcting Currents were no different from the currents that signalled the other five bits of information, it would be possible for the receiver to become mis-synchronized.

In practice the potential for mis-synchronization was not a problem because of the availability of constant human supervision. When the machines were initially turned on they would be allowed to run idle with no data being sent apart from the periodic Correcting Currents for sufficiently long for the armatures to fall into synchronization. If at any point in the day’s work they were to fall out of synchronization the telegraph operator would notice that the text no longer made any sense and could take action to rectify the problem.

In-Band Framing	✓
Transparency	✓
Reliability	✗ (relies on manual intervention)
Locality	✓
Underlying Unit	Physical Signals
Message Size	Fixed (5-bit characters)
Overhead	α = 20%

Table 13. Summary of Framing Properties for Baudot Signalling

2.9.4 RS-232

RS-232 [RS-232-C] uses start and stop bits to mark the boundaries between individual characters. Any larger-scale framing (to group characters into blocks or packets) must be done by a higher level protocol, such as SLIP, PPP, or consistent overhead byte stuffing.

In RS-232 a value of '1' is called a 'mark' and is signalled by a negative voltage on the line. A value of '0' is called a 'space' and is signalled by a zero (or positive) voltage on the line. Every character sent is preceded by a start bit and followed by a stop bit. The start bit is a single bit with value '0' (a space), and the stop bit is a '1' signal (a mark) that lasts for one bit-duration or more. The duration of the stop bit may be longer than one standard bit-time, and need not even be an integral number of bit-times long. Idle times between characters are filled with mark values ('1'), and a long stop bit can be equivalently thought of as a single stop bit followed by some amount of idle time, or a period of idle time may be thought of as being a very long stop bit. When the next character begins, it will begin with a start bit (a space), and it is this transition from mark to space that allows the receiver to detect where the next character begins. This framing is illustrated in Figure 3.

RS-232 start/stop framing is usually used to frame bits into eight-bit bytes. Although RS-232 can also be used to frame bits into five-, six-, seven- or nine-bit units, those modes of operation are less used today now that most computer architectures are based on eight-bit bytes.

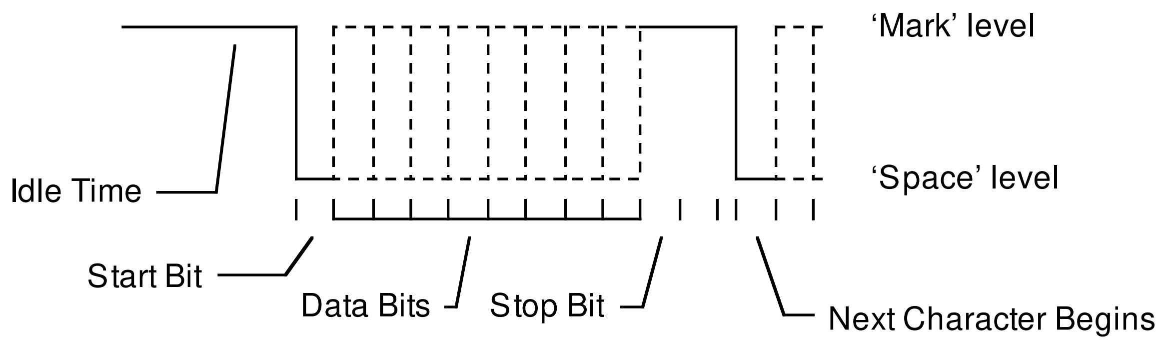


Figure 3. RS-232 Character Framing

RS-232 start/stop framing works well as long as there are no errors, but is not guaranteed to resynchronize quickly after an error. For example, consider sending a sequence consisting of the value 11110000 repeated continuously. With a start bit before and two end bits after each character, the transmitted bit sequence looks like 01111000011. Below is an example sequence beginning with four idle bits, followed by four repetitions of 11110000. Framing bits are shown in a slightly smaller font, in bold type. The spaces between each character are added for clarity — there are no 'spaces' in the transmitted bit-sequence:

1111 **0**11110000**11** **0**11110000**11** **0**11110000**11** **0**11110000**11...**

When all is well, the receiver will detect the transition from one to zero that marks the beginning of each byte. Now suppose that the first zero in the sequence, the first start bit, is lost or corrupted to a one. The receiver will now detect the second transition from one to zero as marking the beginning of a byte, and will incorrectly read the first byte as having value 00011011, instead of 11110000. More significantly, it will also incorrectly read the second byte, and the third, and so on:

1111X1111 **0**00011011**11** **0**00011011**11** **0**00011011**11** **0**00011**...**

With this sequence of data, the receiver will never correctly resynchronize. The receiver will only resynchronize correctly when one of the following happens:

1. The sender sends a character value that contains no transitions from one to zero, such as 00000000, 11111111, or 00001111.

2. The sender pauses for at least eight mark (idle) bits between two characters. This pause allows the receiver's input FIFO to clear, and then it should be able to detect the next one to zero start bit transition correctly.

3. The sender sends at least ten space bits between two characters, indicating a 'break'.

While these events do allow the receiver to resynchronize, it is important to realize that the sender cannot know that the receiver needs to resynchronize unless there is some other mechanism to communicate that information to it. In addition, even if the sending software does know that it needs to insert a pause into the data stream, this is not easy to achieve reliably on many kinds of current computer hardware. When a computer's serial hardware includes output buffering, pauses in the data stream generated by the software may not necessarily be reflected by pauses in the transmitted output bit stream on the RS-232 serial port. To be certain of generating a pause in the output bit stream the software has to cease sending for a grossly pessimistic length of time, to ensure that even a large output buffer will have had time to drain. This use of excessively long pauses is inefficient, and with the data buffers in devices like serial chips and modems getting larger as time goes on, deciding what constitutes a sufficiently conservative length of pause is not easy.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bits
Message Size	Fixed (eight-bit characters)
Overhead	$\alpha = 25\%$

Table 14. Summary of Framing Properties for RS-232

(Best case analysis, using eight bits per character, no parity bit, one stop bit)

2.9.5 SONET

The SONET (Synchronous Optical Network) standard defines a hierarchy of frame sizes and data rates, called the SDH (Synchronous Digital Hierarchy). The basic building block today is usually STS-3 (Synchronous Transport Signal level 3). When used exclusively in an optical context, the STS-3 framing and data rate is also known as OC-3 (Optical Channel level 3). The corresponding compatible European standard is called STM-1 (Synchronous Transport Module level 1) [Sext92].

The structure of a STS-3/OC-3/STM-1 frame is a 2430 byte frame, which is conventionally drawn as nine rows of 270 bytes each. 8000 of these 2430 byte frames per second give the familiar 155.52Mb/sec OC-3 data rate. On each of the rows, the first nine bytes are overhead, and the remaining 261 bytes are payload data. The overhead section of the first row of a frame contains the six-byte framing sequence and three bytes used for other purposes.

The framing sequence is the byte value 11110110 repeated three times followed by the byte value 00101000 repeated three times. It is not signalled via any kind of protocol violation, nor is it protected via any kind of data stuffing procedure to eliminate all accidental occurrences from the packet data. STS-3 is therefore vulnerable to misframing after an error. [Sext92] describes it thus:

> The reframe algorithm is not specified … reframe performance targets are specified in terms of time to detect out of frame (OOF) condition, mean time between false OOF detection in presence of defined error conditions, maximum recovery time from instant when a good signal is available, and maximum probability of false reframing on a good signal.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bits
Message Size	Fixed (2430 byte SONET frames)
Overhead	$\alpha = 3.45\%$

Table 15. Summary of Framing Properties for SONET

2.9.6 Cambridge Ring

Cambridge Ring transports all data in fixed-size ‘minipackets’. The format of a minipacket is: start of packet bit, full/empty bit, monitor passed bit, eight-bit destination address, eight-bit source address, sixteen bits of data, two response bits, and a final parity bit. Thus each minipacket is 38 bits long and contains 16 bits of user data. The ring is populated with as many complete minipackets as will fit into the time delay around the ring; any remaining time not large enough to hold an entire minipacket is unused. The unused bits in the ring are filled with zeroes and form a ‘gap’ which is 0-37 bits in length [Hop86].

Unlike many other networking systems, the Cambridge Ring uses two twisted pairs in tandem to provide a single unidirectional signal. A voltage transition on both pairs at the same time signals a one-bit; a voltage transition on only one of the pairs signals a zero-bit. This encoding enables both pairs to maintain regular transitions (for easy clock recovery) and perfect DC balance, no matter what the data. However, it should be noted that Manchester encoding also maintains regular transitions and perfect DC balance, and although it uses twice the raw signalling rate over the wire, it only uses one twisted pair instead of two, so the total peak signalling rate remains the same.

The Cambridge Ring ‘start of packet bit’ is simply a single bit that has the value one. It is not signalled via any kind of protocol violation, nor (obviously) can it be protected via any kind of bit stuffing procedure to eliminate all one-bits from the packet data. Framing in Cambridge Ring is therefore rather *ad hoc*. [Hop86] describes it thus:

> The framing logic can be in one of two states, **InGap** or **NotInGap**. When a station powers up it may come on in either of these states at random. While a station is in InGap it skips over bits 0 arriving at the input. When a 1 arrives it takes this to be the start of the slot train and becomes NotInGap. In this state it counts 38 bits from the 1 it received and looks at the next bit. If this is a 1 it remains in the NotInGap state and waits another 38 bits. If it is a 0 the station goes to the InGap state. Eventually, this process will get the station into synchronisation with the ring.

This description of the algorithm assumes that this relatively slow synchronization process only has to happen once, upon power-up, after which the ring operates nearly error-free. [Hop86] goes on to describe what happens after 'a serious error occurs':

> Because stations will not synchronise immediately to these slot trains they will probably become mutilated as they pass round the ring and stations try to correct parity bits which are not really parity bits. Stations may even try to transmit in the slots.

While Cambridge Ring framing may have worked well in the low-error environment it was designed for, it does not have great applicability as a good general-purpose framing algorithm.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bits
Message Size	Fixed (38 bit minipackets)
Overhead	α = 137.5%*

Table 16. Summary of Framing Properties for Cambridge Ring

> * If you take into account the fact that Cambrige ring actually used *two* 10Mb/s twisted pairs to carry a single 10Mb/s signal, the effective overhead is 375%. If you further take into account that a typical ring might be 100 bits long, which is enough to contain only two 38-bit minipackets and a 24-bit gap, the effective overhead is 525%. To put this information into perspective, using two twisted pairs carrying a combined 20Mb/s Cambridge Ring achieves a raw throughput of no more than 3.2Mb/s, before taking other higher-layer headers and overhead into account.

2.9.7 RATP

RATP (Reliable Asynchronous Transfer Protocol) [RFC916] [RFC935] uses a packet format that begins with the ASCII SOH (Start of Heading) character (0x01) followed by a one-byte length count. Any data values can appear in the payload, and no data stuffing is performed. Since character value 0x01 is allowed to appear in the payload of the packet, the protocol is not guaranteed to resynchronize rapidly after an error.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bytes
Message Size	Variable, up to 255 bytes
Overhead	$\alpha = 0\%$ $k = 16$ bits

Table 17. Summary of Framing Properties for RATP

2.9.8 Digital Equipment Corporation DDCMP

DDCMP (Digital Data Communications Message Protocol) [Held91] uses a packet format that is similar to RATP. A packet begins with two ASCII SYN characters (0x16) followed by a one-byte type, followed by a two-byte count/flags field (14 bit count, two bit flags) three more header bytes, a two-byte header CRC, the packet data, and a final two-byte data CRC.

As with RATP, no data stuffing is performed and since ASCII SYN characters are allowed to appear in the payload of the packet, the protocol is not guaranteed to resynchronize after an error.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bytes
Message Size	Variable, up to 16383 bytes
Overhead	$\alpha = 0\%$ $k = 96$ bits

Table 18. Summary of Framing Properties for DDCMP

2.9.9 XMODEM (The Christensen Protocol)

XMODEM [For88] was written by Ward Christensen in 1977. Like DDCMP, XMODEM uses a marker character to indicate the start of each block. The chosen marker character is the ASCII SOH (Start of heading) character (0x01). The SOH is followed by the block number, then the ones-complement of the block number, then a fixed size block of 128 data bytes, and then finally a one-byte checksum on the end. Since ASCII SOH characters are allowed to appear in the user data and no stuffing is performed, the protocol is not guaranteed to resynchronize after an error.

Since XMODEM is designed to be used by a human operator, it does have a kind of out-of-band recovery mechanism. If modem line noise causes the protocol to go horribly wrong, then the user simply cancels the transfer and waits for the software at the other end to time out. Since the data bytes contained in the file being transferred can never be encoded as 'a really long pause until the software at the other end times out' this pause is effectively an out-of-band signal which cannot occur in any valid data transfer. It is a far cruder signal than the brief pause that precedes an Ethernet packet, but it has the same effect of resynchronizing the sender and receiver.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bytes
Message Size	Fixed (128 bytes)
Overhead	$\alpha = 3.125\%$

Table 19. Summary of Framing Properties for XMODEM

2.9.10 YMODEM

YMODEM [For88] is a successor to XMODEM, and adds a number of small improvements. YMODEM adds the option of sending 1024-byte blocks instead of 128-byte blocks. It adds the option of a sixteen-bit CRC instead of an eight-bit checksum. It adds user interface improvements like the ability to transfer multiple files in a single batch transfer and the ability to specify the filename(s) of the file(s) being transferred. Despite these improvements, the basic design philosophy of the protocol is otherwise unchanged. It still uses an initial marker character, followed by the block number, then the ones-complement of the block number, then a block of (128 or 1024) data bytes, and then a (one or two byte) check value on the end, and it still performs no data stuffing. The size of the block being sent is indicated by the initial marker character. If it is ASCII SOH, it is an XMODEM-style 128-byte block, and if it is an ASCII STX (Start of text) character (0x02), it is a 1024-byte block.

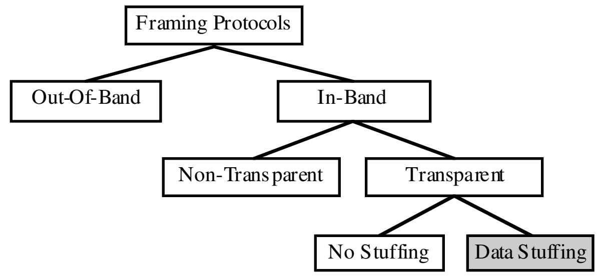
Although YMODEM was supposed to be an improvement over XMODEM, the way it is extended to support two block sizes actually makes YMODEM even more prone to mis-synchronization after an error than XMODEM is, because now there are two valid initial marker characters to look for instead of just one. Given data containing uniformly distributed values, when using XMODEM the probability that any particular byte will accidentally be the marker character is $^1/_{256}$. When using YMODEM, the probability that any particular byte will accidentally be a marker character is $^2/_{256}$.

In-Band Framing	✓
Transparency	✓
Reliability	✗
Locality	✓
Underlying Unit	Bytes
Message Size	Choice of two fixed sizes (128/1024 bytes)
Overhead	α = 0.39% to 3.91%*

Table 20. Summary of Framing Properties for YMODEM

* The overhead of 0.39% is achieved when using a one-byte checksum and 1024-byte blocks. The overhead is a marker character, two block number characters, and a single checksum character (four characters total) per 1024 bytes of user data. The overhead of 3.91% is achieved when using a two-byte CRC and 128-byte blocks. The overhead is a marker character, two block number characters, and two CRC characters (five characters total) per 1024 bytes of user data.

2.10 Transparent In-Band Framing Protocols With Data Stuffing



This section describes IBM BISYNC, Kermit, ZMODEM, SLIP, AX.25, PPP, Gill's Protocol, and HDLC.

2.10.1 IBM BISYNC

The IBM Binary Synchronous protocol [Mil87] can operate in non-transparent and transparent modes.

In non-transparent mode, the start of the block of data is indicated by an ASCII STX (Start of text) character (0x02) and the end by an ASCII ETX (End of text) character (0x03). Obviously this framing will not work if the data itself contains an ETX character.

To remedy this problem sending raw 'binary' data, IBM BISYNC also has a transparent mode which makes use of the ASCII DLE (Data Link Escape) character (0x10). Instead of using just STX and ETX, the start of the block of data is indicated by the sequence [DLE, STX] and the end by a [DLE, ETX] sequence. An ETX character appearing on its own has no special meaning, so it can be safely used in the packet data. If the packet data contains a DLE character, then two DLE characters are transmitted to represent that single DLE character. When the receiver sees two DLE characters, it interprets it as representing a single DLE character in the data. When the receiver sees [DLE, ETX], it interprets it as indicating the end of the packet.

IBM BISYNC was one of the first protocols to incorporate features that are now common in communication protocols. It is able to transmit reliably any data value, and to achieve that transparency it uses data transformation (variously called character prefixing, character escaping, or byte stuffing), not out-of-band signalling. Just as with the well-known protocols SLIP and PPP (described below), that transformation can double the size of the data in the worst case (a packet entirely containing DLE characters). Just as with SLIP and PPP, recovery after error is guaranteed to be rapid, no matter what data the packet payloads carry.

Unlike SLIP and PPP, IBM BISYNC does not use its escape character (DLE) consistently. When placed in front of another DLE character, it indicates that the following character is to be treated as data. When placed in front of any other character, it indicates that the following character is to be treated as a control character.

The use of the escape character in this way means that IBM BISYNC fails to meet the strict definition of framing locality as required in this dissertation. This dissertation defines the property *locality* to mean that no combination of insertion, deletion, or replacement errors preceeding the initial framing marker, nor following the final framing marker, can prevent the receiver from receiving and decoding the packet correctly. IBM BISYNC fails to meet that requirement because an erroneous DLE character received just before the initial framing marker will make the receiver ignore the following DLE character and consequently the receiver will fail to recognize the DLE ETX mark correctly, causing the next packet to be lost as well.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✗
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 0.39\%$ $\beta = 100\%$ $k = 32$ bits

Table 21. Summary of Framing Properties for IBM BISYNC

2.10.2 Kermit

Kermit was designed to allow file transfer between Columbia University's central time-sharing system and personal computers connected over serial lines [Cru78]. The designers could not assume that the communications path between the time-sharing system and a personal computer would reliably transmit characters other than printable seven-bit ASCII (0x20-0x7F) and the ASCII SOH (Start of heading) character (0x01). For this reason, the Kermit protocol uses an escape prefix for all ASCII control characters and all characters that have the top-bit set. Control characters (ASCII codes less than 0x20, and 0x7F) have bit six inverted (exclusive OR with 0x40) and are prefixed with the '#' character. Thus character code 0x01 is encoded as the two-character sequence '#A'. ASCII characters that have the top-bit set have bit seven inverted (exclusive OR with 0x80) and are prefixed with the '&' character. Thus character code 0xC1 is encoded as the two-character sequence '&A'. If inverting bit seven results in a control character as described above, then the first rule is also applied to the resulting character. Thus character code 0x81 is encoded as the three-character sequence '&#A'. This gives Kermit a worst-case expansion of three to one for pathological data.

Even in the average case for uniformly distributed data, Kermit is not very efficient. All characters that have the top bit set (half of them, on average) add one prefix byte to the output. This transformation reduces us to a space of 128 valid characters, and 35 of those (characters 0x00 to 0x1F, '#', '&' and 0x7F) also add one prefix byte to the output. This means that, with its default encoding rules, Kermit expands uniformly distributed data by:

$$\frac{1}{2} + \frac{32+3}{128} = 77.34\%$$

Of course, Kermit was not designed for transferring uniformly distributed data. It was designed for transferring printable ASCII text where the only non-printable character likely to appear was the carriage return at the end of each line.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 77.34\%$ $\beta = 200\%$ $k = 8$ bits

Table 22. Summary of Framing Properties for Kermit

2.10.3 ZMODEM

ZMODEM [For96] uses a reserved value, the ASCII CAN (Cancel) character (0x18), to mark the end of a packet, and uses character escape sequences to replace any occurrences of ASCII CAN in the body of the data. In addition to escaping ASCII CAN, ZMODEM also escapes other characters that can cause problems in dial-up networks, such as the X-On and X-Off flow-control characters.

ZMODEM has the advantage that it provides rapid recovery after error, and the disadvantage that in the worst case, where every data byte has to be replaced with a two-character escape sequence, it can double the size of the data.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 0.78\%$* $\beta = 100\%$ $k = 8$ bits

Table 23. Summary of Framing Properties for ZMODEM

* The average overhead of 0.78% assumes that only two characters, CAN and the escape character itself, are replaced by two-byte escape sequences. When more characters are escaped, the average overhead is higher.

2.10.4 SLIP and AX.25

Serial Line IP [RFC1055] is a very simple and straightforward encoding of IP [RFC791] packets, which probably accounts for its widespread use today.

Like BISYNC and ZMODEM, SLIP uses a reserved end-of-packet character, and uses an escape code to transform the data to eliminate that end-of-packet character.

In SLIP, the end-of-packet character is octal 300 (0xC0). Anywhere that octal 300 occurs in the packet data, it is replaced with the sequence octal [333, 334], and anywhere that octal 333 occurs in the packet data, it is replaced with the sequence octal [333, 335]. Octal 300 is called the SLIP END character, and Octal 333 is called the SLIP ESC character.

Like BISYNC and ZMODEM, SLIP has the advantage that it provides rapid recovery after error, and the disadvantage that in the worst case, where every data byte has to be replaced with an two-character escape sequence, it can double the size of the data.

AX.25 [ARRL84] is very similar to SLIP. AX.25 adds a link-layer header to the packet, but the byte stuffing algorithm used is identical, so in terms of reliability and overhead variability it has the same advantages and drawbacks as SLIP.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 0.78\%$ $\beta = 100\%$ $k = 8$ bits

Table 24. Summary of Framing Properties for SLIP/AX.25

2.10.5 PPP

PPP, the Point-To-Point Protocol [RFC1661] improves on SLIP by providing support for sending other protocols in addition to IP. PPP also provides support for encryption, authentication, and many other services, including the provision for optional negotiation of future protocol extensions. However, in the context of this dissertation, the relevant aspect of PPP is how it frames its packets, as specified in 'PPP in HDLC-like Framing' [RFC1662]. While PPP's Link Control Protocol, described in 'The Point-to-Point Protocol (PPP)' [RFC1661] allows implementations of PPP to negotiate with their peers regarding the use of new optional features, when this dissertation refers to 'PPP byte stuffing', it is referring to this conventional implementation of PPP framing, which is also referred to variously as 'AHDLC' (when used on asynchronous links) or 'octet synchronous' (when used on synchronous links), not some future PPP implementation augmented to use a new framing mechanism (such as COBS).

In its simplest form of operation, the standard PPP framing technique is identical to SLIP's technique with different character values used in place of SLIP ESC and SLIP END. PPP's end-of-packet character is ASCII '~', (0x7E), which in binary is 01111110 (the same as the HDLC frame delimiter). PPP's escape character is 0x7D. Anywhere that 0x7E occurs in the packet data, it is replaced with the sequence [0x7D, 0x5E], and anywhere that 0x7D occurs in the packet data, it is replaced with the sequence [0x7D, 0x5D]. However, instead of just defining these two specific substitutions as special cases, PPP goes further to define a general rule of how characters are escaped: To escape a character that should not appear in the transmitted packet data, PPP prefixes the character with 0x7D and inverts bit five of the character (i.e. it is exclusive or'd with 0x20). 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.

Unlike SLIP's specific substitutions, PPP's more general definition of character escaping allows it to also escape other characters that may need to be eliminated from the transmitted packet data for other reasons. For example, if the packet is to be transmitted over a non-transparent dial-up network it may be necessary to eliminate the X-On and X-Off flow-control characters. In extreme cases, PPP is able to eliminate all ASCII control characters (0x00-0x1F) from the transmitted packet data, by prefixing all of them with 0x7D and exclusive or'ing with 0x20 to map those characters into the range 0x20-0x3F.

It is equally possible to do this kind of multiple value elimination using a COBS-type algorithm, but the subject of this dissertation 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.

Like BISYNC, ZMODEM and SLIP, PPP has the advantage that it provides rapid recovery after error, and the disadvantage that in the worst case, where every data byte has to be replaced with an two-character escape sequence, it can double the size of the data. This overhead is much worse than HDLC's worst case of only 20%, but PPP byte stuffing has the advantage that it can be implemented efficiently in software quite easily. HDLC is described below.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bytes
Message Size	Unlimited
Overhead	$\alpha = 0.78\%$ $\beta = 100\%$ $k = 8$ bits

Table 25. Summary of Framing Properties for PPP

2.10.6 John Gill's XWindow Modem Protocol

In 1991, John Gill at Stanford University devised a protocol for framing XWindow protocol units over a modem connection or a 56kb/sec serial line, for the local Digital Equipment Corporation office in Palo Alto [Gill96].

John Gill's protocol uses byte value 0xFF as the framing character; hence it must eliminate all occurrences of 0xFF from the packet payload. Gill does this elimination by beginning each packet with a 'pointer byte'. The pointer byte gives the address within the packet of the first occurrence of 0xFF; if there is no occurrence, this pointer is zero. The address pointed to by the first byte contains either a pointer to the next occurrence of 0xFF or a zero to mark the end of the chain, and so on. One of the nice features of this scheme is that the escape bytes can be written back into their locations in the buffer without moving data around; this is useful when data has been copied in by some dumb I/O device [Gill96].

This data transformation is elegant and easy to implement efficiently in software, but its drawback is that it cannot encode packets of arbitrary size. The pointer bytes each have to fit into one byte, and byte value 0xFF is reserved as the framing marker, so only values 0x00 to 0xFE can be used. This limitation means that the protocol cannot directly encode packets longer than 254 bytes (not including the initial pointer byte, which is added by the encoding).

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bytes
Message Size	Variable, up to 254 bytes
Overhead	$\alpha = 0\%$ $k = 8$ bits

Table 26. Summary of Framing Properties for Gill's Protocol

2.10.7 HDLC

Many of the protocols above operate on eight-bit bytes and assume that some underlying hardware or software protocol will interpret the raw electrical signal, detect the byte boundaries correctly, and present convenient eight-bit bytes for the packet framing protocol to operate on. In contrast, HDLC [ECMA-40] is a bit stuffing scheme that operates on the raw stream of binary digits. It uses the binary sequence 01111110 as the frame delimiter. This delimiter appears immediately before and immediately after every frame, and there can be no pause within the data of a frame. If two frames are sent back-to-back, then the ending delimiter of one frame may also be the starting delimiter of the next. If there is an extended pause between frames, then a continuous stream of frame delimiters may be sent to fill the idle time.

A binary sequence of seven or more ones in a row is the abort sequence. A binary sequence of fifteen or more ones in a row is the idle sequence, and may be used instead of a continuous stream of frame delimiters. The idle sequence also implies abort; the idle sequence may be used in spaces between packets, but it may not be used to generate a pause within a packet.

For HDLC to work properly, the frame delimiter, abort, and idle patterns must not occur anywhere within the data of a frame. To eliminate these patterns from the data, HDLC uses the following transformation: Whenever the transmitter observes five ones in a row, it inserts a zero immediately following. This procedure eliminates the possibility of six or more ones ever occurring inadvertently in the data. The receiver performs the reverse process: After observing five ones in a row, it examines the next binary digit. If it is a zero, the digit is deleted and reception of the frame continues. If it is a one, then the receiver knows it is seeing a special control sequence and it examines the following digit after that to determine what action to take. If it is a zero then this is a frame delimiter, so the complete frame is handed up to the next protocol layer. If it is a one then it is the abort sequence, and any current frame data that had been received up to that point is discarded.

HDLC has many good properties. After an error it always resynchronizes immediately upon reception of the next frame delimiter, and it is transparent to higher layers. Even in the worst case (frame data that is entirely binary ones) HDLC's insertion of extra zeroes ('bit stuffing' or 'zero insertion') adds only 20% to the transmitted size of the data.

The drawback of HDLC is that it requires bit-by-bit processing of the data. This kind of bit-level manipulation is easy to implement in the hardware of a serial transmitter, but it 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 the output byte-stream is not kept 'in-phase' with the input byte-stream, making it difficult to take good advantage of the processor's ability to operate on bytes or words at a time. A hardware implementation of HDLC typically reads a bit at a time, and writes one or two bits in response, depending on whether a stuffing event occurs. An efficient software HDLC implementation could be written that reads eight bits at a time, but it would then have to write eight, nine, or ten bits in response, depending on whether zero, one, or two stuffing events occur within those eight bits. Writing nine or ten bits to memory is not an operation that is usually supported efficiently by memory systems, and even if it were, it would mean that the following write would not be aligned to an eight-bit boundary. An algorithm could be devised that wrote only eight bits, and kept any additional bits stored in the form of internal state in the finite state machine until sufficient stored bits had accumulated for it to write out an entire byte of stored state. This approach is possible but it would result in a finite state machine with 251 internal states, and in each state, 256 different one- or two-byte outputs and next-state transitions depending on the value of the input. Such a finite state machine would not be very amenable to efficient compact software implementation. It would require either a large lookup table of state transitions, which would not be very compact, or it would require a smaller state table in conjunction with shifting and masking of bits in registers, which would not be very efficient.

In-Band Framing	✓
Transparency	✓
Reliability	✓
Locality	✓
Underlying Unit	Bits
Message Size	Unlimited
Overhead	$\alpha = 1.56\%$ $\beta = 20\%$ $k = 8$ bits

Table 27. Summary of Framing Properties for HDLC

2.11 Summary of Framing Protocols

The following table summarizes the properties of all the protocols described above:

	In-Band	Transp-arency	Relia-bility	Local-ity	Underly-ing Unit	Message Size	α (%)	β (%)	*k* (bits)
Ethernet		✓	✓	✓	Physical	Unlimited	100		64
Token Ring		✓	✓	✓	Physical	Unlimited	100		8
FDDI	✓	✓	✓	✓	Physical	Unlimited	25		280
Teletext		✓	✓	✓	Physical	40 chars	20		
ISDN ‘S’		✓	✓	✓	Physical	32 bits	50		
U.S. ISDN ‘U’	✓	✓		✓	Quats	192 bits	25		
Non-U.S. ISDN ‘U’	✓	✓		✓	Ternary	192 bits	6.8		
MPEG	✓				Bytes	Unlimited	0		32
T-1 D4/AMI	✓				Bits	192 bits	0.52		
T-1 ESF/B8ZS	✓	✓			Bits	192 bits	0.52		
Verilink		✓			Bytes	≤127 Bytes	0		1
Baudot	✓	✓		✓	Physical	5 bits	20		
RS-232	✓	✓		✓	Bits	8 bits	25		
SONET	✓	✓		✓	Bits	2430 bits	3.45		
Cambridge Ring	✓	✓		✓	Bits	38 bits	137.5		
RATP	✓	✓		✓	Bytes	≤255 Bytes	0		16
DDCMP	✓	✓		✓	Bytes	≤16383 B	0		96
XMODEM	✓	✓		✓	Bytes	128 Bytes	3.125		
YMODEM	✓	✓		✓	Bytes	128/1024 B	.39-3.9		
IBM BISYNC	✓	✓	✓		Bytes	Unlimited	0.39	100	32
Kermit	✓	✓	✓	✓	Bytes	Unlimited	77.34	200	8
ZMODEM	✓	✓	✓	✓	Bytes	Unlimited	0.78	100	8
SLIP	✓	✓	✓	✓	Bytes	Unlimited	0.78	100	8
PPP	✓	✓	✓	✓	Bytes	Unlimited	0.78	100	8
Gill’s Protocol	✓	✓	✓	✓	Bytes	≤254 Bytes	0		8
HDLC	✓	✓	✓	✓	Bits	Unlimited	1.56	20	8

Table 28. Summary of Table for all Framing Protocols

The conclusion reached in this dissertation, that many current framing protocols are not reliable, may seem unduly pessimistic, considering that they are apparently used very widely without trouble. However, the truth is that although the problems are not widely publicized, they do exist. One example is that unfortunate patterns in PPP data frequently disrupt SONET links, “even when interleaved with other traffic.” [Tay97]

The only protocols that meet all four requirements of in-band framing, transparency, reliability and locality are FDDI, Kermit, ZMODEM, SLIP, PPP, Gill’s protocol and HDLC.

Of these protocols, FDDI has a fixed overhead for all packets of 25% plus 280 bits, and Kermit has an average overhead of 77.34%. These are both very much higher than COBS’s overhead.

ZMODEM, SLIP and PPP are all two-for-one substitution protocols and have average overheads under 1%. Although they have low average overheads, their behaviour is very variable. Sometimes they add no overhead at all, and sometimes they double the size of the packet. These three protocols use different byte values for their escape and framing characters, but otherwise their two-for-one substitution behaviour is identical.

Gill’s protocol has very low overhead, adding under 0.4% overhead for 254-byte packets, but cannot encode any packet larger than this. While very useful for X Windows packets, this limitation makes it unsuitable for use as a general purpose framing protocol.

HDLC is in many ways the best of the protocols. It adds an average of only 1.56% and guarantees at worst to add at most than 20% overhead, a guarantee that the two-for-one substitution protocols cannot match. However, HDLC requires bit-level processing of the data which can be hard to do efficiently in software (see Section 2.10.7); for this reason many protocol designers and implementers have shied away from it, preferring the easier byte-oriented approach.

In contrast to all the previous protocols, Consistent Overhead Byte Stuffing:

- meets all four requirements of in-band framing, transparency, reliability and locality
- can encode arbitrary-sized frames
- has low average overhead (0.23%)
- has low worst-case overhead (0.4%)

This dissertation makes a detailed comparison between COBS and two of the other protocols, PPP and HDLC. PPP was chosen as a representative example of one of the two-for-one substitution protocols. Any of the two-for-one substitution protocols could have been selected, but PPP was chosen because it is very widely used today, and in many people's minds PPP is the 'modern' replacement for SLIP. HDLC was chosen as the other protocol with which to compare COBS, because it is the only protocol that has an even remotely competitive worst-case overhead bound.

Chapter Three

Algorithms

This chapter begins by describing the data encoding format used by COBS and the procedure for converting packet data to and from that encoded format. It then describes some of the properties and implications of using COBS encoding, and describes a variant of COBS called COBS/ZPE in which has slightly different properties.

3.1 COBS Syntax and Semantics

COBS performs a reversible transformation on a data packet to eliminate a single byte value from it. Once eliminated from the data, that byte value can then safely be used as the framing marker without risk of ambiguity.

For the description that follows, zero was chosen as the framing value to be eliminated. In practice zero is a good value to choose for a real-world implementation because zeroes are common in binary computer data, and COBS performs marginally better when it has many bytes to eliminate. However, elimination of some value other than zero can be achieved with only trivial changes to the algorithm (Section 6.1).

COBS first takes its input data and logically appends a single zero byte. (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.)

COBS then locates all the zero bytes in the packet (including the added one), and divides the packet data into one or more *zero-terminated chunks* so that every chunk ends with one of these zero bytes, and contains no other zeroes — every chunk contains exactly one zero byte, and that zero is always the last byte of the chunk. Every chunk is therefore a sequence of zero or more non-zero bytes, followed by a single zero byte. A chunk may be as short as one byte (i.e. a chunk containing just a solitary zero byte) or when the packet contains no zeroes, a chunk may be as long as an entire packet.

COBS encodes each zero-terminated chunk using one or more variable length *COBS code blocks*. Chunks of 254 bytes or fewer are encoded as a single COBS code block. Chunks longer than 254 bytes are encoded using two or more code blocks, as described later in this section. After all the chunks which make up the packet have been encoded using COBS code blocks, the entire resulting aggregate block of data is completely free of zero bytes, so zero bytes can then be placed around the encoded packet to mark clearly where it begins and ends.

A COBS code block consists of a single code byte, followed by zero or more data bytes. The number of data bytes is determined by the code byte. Figure 4 shows some examples of valid COBS code blocks and the corresponding zero-terminated data chunks they represent.

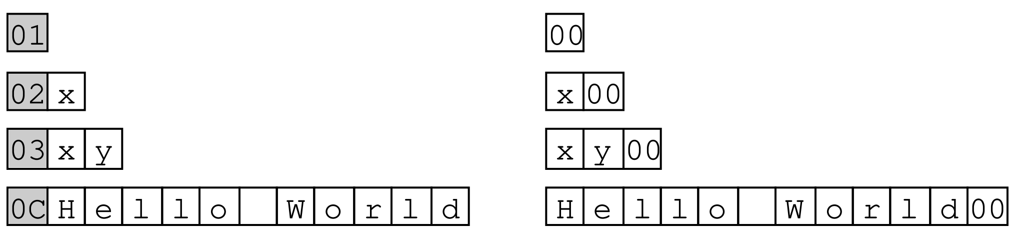


Figure 4. Example Code Blocks and the Data Chunks they Represent

Each COBS code block begins with a single code byte (shown shaded), followed by zero or more data bytes.

For codes 0x01 to 0xFE, 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 byte*. The zero byte is implicit — it is not actually contained within the sequence of data bytes in the code block.

These code blocks encode data without adding any overhead. Each code block begins with one code byte followed by n data bytes, and represents n data bytes followed by one trailing zero byte. Thus the code block adds no overhead to the data: an n+1 byte chunk is encoded using a 1+n byte code block.

These basic codes are suitable for encoding zero-terminated chunks up to 254 bytes in length, but the zero-terminated chunks that make up a packet may be longer that that. To cope with these chunks, code 0xFF is defined slightly differently. Code 0xFF represents the sequence of 254 data bytes contained within the code block, *without any implicit zero*. Defining the maximum length code (0xFF) as an exception this way allows us to encode chunks that are too long to be encoded as a single code block. Chunks longer than 254 bytes are encoded using one or more of these special maximum length code blocks, followed by a single normal code block, as shown in Figure 5. Each special maximum length code block has no implied trailing zero, but the final (normal) code block does include an implied trailing zero at the end, so this sequence of code blocks correctly encodes the required long chunk of non-zero data with a single zero byte at the end. Unlike the other code blocks, this maximum length code block does add some overhead to the encoded data. 254 bytes of packet data are encoded using 255 bytes, an overhead of just under 0.4%.

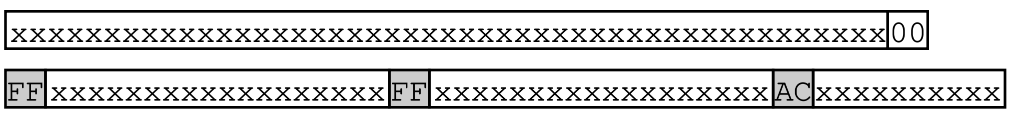


Figure 5. Encoding Oversized Chunks

In this example, a 680-byte zero-terminated chunk (679 non-zero bytes and one zero) is encoded using three COBS code blocks. The first two are special maximum-length (254 data bytes, no implied zero) COBS code blocks, and the final code block is a standard 172-byte (171 data bytes and an implied zero) COBS code block. Thus 680 bytes of user data is encoded using 682 bytes, an overhead of just under 0.3%.

The meanings of the various COBS code values are summarized in Table 29.

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 [0,0xFF] and produces an output consisting of characters only in the range [1,0xFF]. 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	no data bytes	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 29. Code Values used by Consistent Overhead Byte Stuffing

For codes 0x01 to 0xFE, a code byte value of n represents a series of n bytes where the final byte (and only the final byte) a zero. The (n–1) non-zero bytes are placed immediately after the code byte; the trailing zero is implicit. The one exception is code byte 0xFF, which represents a series of (n–1) bytes without any trailing zero.

3.2 COBS Encoding Procedure

The job of the COBS encoder is to translate the raw packet data into a series of COBS code blocks. Breaking the packet into zero-terminated chunks as described above is a useful way to understand the logic of COBS encoding, but in practice it is more efficient to write an encoder that translates from packet data directly to code blocks, without going through an intermediate data format.

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. The zero is then skipped and this process is repeated until all the bytes of the packet (including the final zero which is logically appended to every packet) have been encoded. Figure 6 shows an example of packet encoding.

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 6. Example Encoding

> The input data is shown with a phantom zero logically appended to the end. The encoded form is shown below the input, 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).

The implementation of COBS is very simple. The Appendix gives complete C source code listings to perform both COBS encoding and decoding.

There is a small optimization that may be performed in one particular situation. The reason for logically appending a zero to every packet before encoding is that all but one of the code blocks represent a block of data that ends with an implied zero. If the data to be encoded did not actually end with a zero it might not be possible to encode it properly using COBS code blocks. However, there is one code block, code block 0xFF, that does not end with an implied zero. If by chance the packet happens naturally to end with a maximum length 0xFF code block then there is no need to logically append a zero in order to make it encodable. 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. In practice, the gain achieved by this optimization is very slight, but it is useful from a theoretical standpoint. It simplifies analysis of the algorithm's behaviour by allowing us to say that COBS encoding adds "no more than one byte of overhead for every 254 bytes of packet data." Without this optimization, that statement would not be true for packets containing an exact multiple of 254 bytes and no zeroes; these packets would incur one extra byte of overhead.

3.3 Behaviour of COBS

COBS encoding has low overhead (on average 0.23% of the packet size, rounded up to a whole number of bytes) and furthermore, for packets of any given length, the amount of overhead is almost constant, regardless of the packet contents. This section offers an intuitive understanding of why the overhead of COBS is so uniform, in marked contrast to the overhead of PPP and other two-for-one byte stuffing schemes, which are extremely variable.

COBS encoding has low overhead because, in most cases, the size of each code block is exactly the same as the size of the data sequence it encodes. For all of the code blocks 0x01 to 0xFE, the code block contains one code byte and (n–1) non-zero data bytes. The data represented by that code block is the (n–1) data bytes followed by a single zero byte at the end. Hence n input bytes are encoded using exactly n output bytes, so the output size is the same as the input size. For code block 0xFF, there is no implied zero byte on the end, so for those code blocks 254 input bytes are encoded using 255 output bytes, an overhead of just under 0.4%. In the best case where no maximum length 0xFF code blocks are required to encode the packet, COBS encoding adds no overhead apart from the trailing zero which is logically appended to every packet, giving an overhead of just one byte, regardless of how large the packet is.

COBS encoding overhead has very low variability compared to other byte stuffing algorithms like PPP. To see why this is so we need to understand how overhead is generated in byte stuffing algorithms. For any byte stuffing algorithm like PPP or COBS, each byte in the packet either generates no overhead, or it generates some fixed amount x. In a given packet of length n bytes, if the proportion of bytes that are overhead-generating is p, then the total overhead for the packet will be npx bytes. For each byte stuffing algorithm, x has some fixed value, but the values of n and p are determined by the size of the packet being encoded and the nature of the data in that packet.

For PPP, overhead is incurred whenever a byte has one of the reserved values 0x7D or 0x7E. On average for uniformly distributed data, one byte in 128 is a reserved value. This occurrence is relatively rare — less than 0.8% of the bytes — but when it does occur the cost is relatively high — an encoding overhead of 100% for that particular byte. All the other byte values — the vast majority — incur no overhead. Thus for PPP x=1 and, for the average packet, $p={}^{1}/_{128}$.

For COBS, the behaviour has the opposite characteristics. Overhead is not the rare case; it is the common case. Overhead is not generated by occurrences of the reserved value, it is generated by the large majority of bytes that are *not* the reserved value zero: If more than 254 bytes are encountered without finding a single zero, then one byte of overhead is generated. Fortunately, although overhead is the common case, the one byte of overhead generated is amortized over 254 bytes, making the cost per-byte very small. The rare event of encountering a zero can be viewed as an occasional lucky bonus: both the zero and the series of non-zero bytes that preceded it are then encoded with no overhead at all. Thus for COBS the proportion of bytes that are overhead-generating is high — usually the majority of the bytes in the packet — but the amount of overhead each one generates, x, is only $^{1}/_{254}$.

Because of these two opposite kinds of behaviour, PPP and COBS generate overhead in opposite ways. The encoding overhead of PPP can be characterized as the sum of a series of high cost low probability events, whereas the overhead of COBS can be characterized as a series of high probability low cost events. For PPP to encode a packet with low overhead, it is dependent on the proportion of overhead-generating bytes in the packet being small. In the pathological case where every byte in the packet is an overhead-generating byte, p=1 and the size of the packet doubles. In contrast, COBS already assumes that most bytes are overhead-generating bytes, so the worst possible case where every byte is overhead-generating is hardly any different from the average case. For the same reason, the best case for COBS is also hardly any better than the average case. Because x is so small, the final encoded size of the data is very insensitive to the proportion p of bytes that are overhead-generating.

A related observation is that the worst cases for PPP and for COBS occur in opposite circumstances. The rare case where every byte is a reserved value is the worst case for PPP and results in the maximum overhead. For COBS the rare case where every byte is a reserved byte is the best case and results in the minimum overhead.

The overhead incurred by encoding a packet using COBS is at least one byte, and at most one byte for every 254 bytes of packet data. This is good for large packets, but has an unfortunate side-effect for small packets. Every packet 254 bytes or smaller always incurs exactly one byte of overhead, no more, no less.

I regard one byte of overhead for small IP packets a small price to pay in exchange for vastly more predictable overhead and the consequent significant performance gains we get from the ability to send much larger IP packets. However, there could be circumstances where any cost, however small, is unacceptable. To address this concern this dissertation also presents 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 experiments on real-world data (see Chapter 5) I observed not only that zero is a common value in Internet traffic, but furthermore that adjacent pairs of zeros are also very common, especially in the headers of small TCP/IP packets. To take advantage of this property of real-world traffic, I created a variant of COBS where the maximum encodable sequence length is slightly shortened, freeing some of the high-numbered codes for other uses. These codes are reassigned to indicate sequences ending with a *pair* of implicit zeroes. Any reduction in the maximum sequence length increases the worst-case overhead, so the maximum length should not be reduced by too much. Empirically I found that reassigning 31 codes gives good performance without sacrificing too much of the good worst-case bound that makes COBS useful.

Code	Followed by	Meaning
0x00	(not applicable)	(not allowed)
0x01-0xDF	(*n*–1) data bytes	The (*n*–1) data bytes, followed by a single zero
0xE0	223 data bytes	The 223 data bytes, **not** followed by a zero
0xE1-0xFF	(*n*–225) data bytes	The (*n*–225) data bytes, followed by a **pair** of zero bytes

Table 30.
Code Values used by Consistent Overhead Byte Stuffing with Zero-Pair Elimination

Codes 0x01 to 0xDF have the same meaning as in basic COBS, but codes 0xE0 to 0xFF are reassigned to have new meanings.

In COBS/ZPE 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 change in code assignments gives COBS/ZPE a worst-case overhead of one byte in 223, or about 0.45%. Codes 0xE1 to 0xFF encode sequences that end with an implicit pair of zeroes, containing, respectively, 0 to 30 non-zero data bytes. These new code meanings are summarized in Table 30.

Reassigning some of the codes to indicate sequences ending with a *pair* of implicit zeroes has the good property that now some of the code blocks are *smaller* than the data they encode, which helps mitigate the one-byte overhead that COBS adds. Figure 7 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 Chapter 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.

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 7. Example Encoding with Zero-Pair Elimination

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

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 zeroes between fields. These padding zeroes waste precious bandwidth on slow wireless links. Using COBS/ZPE as the link-layer encoding for these slower links can help to mitigate this effect by encoding these patterns more efficiently. This increased efficiency for zero pairs makes it more attractive to use the same packet formats regardless of network speed, instead of having to design different application-layer protocol formats for different speeds of network.

The reason that reassignment of 31 codes was found to be a sensible cut-off point for this particular performance evaluation is that runs of 30 non-zero bytes followed by a zero pair were found to be common in our IPv4 traces. One factor that contributed to the frequent occurrence of this pattern is that IPv4 packet headers often contain a zero at offset 7 in the IP packet (the low byte of the fragment offset) and the TCP header often contains a pair of zeroes at offset 38 in the IP packet (the urgent pointer). This means that at offset 8 in the packet there is often a run of 30 non-zero bytes, followed by a zero pair. For other protocols and traffic mixes the 'optimum' cut-off point may be different, but it should be remembered that the benefit of COBS (and COBS/ZPE) encoding is that it is extremely insensitive to the nature of the data being encoded. The penalty for picking the 'wrong' cut-off point is very slight.

Although for certain packets COBS/ZPE does result in some reduction in size, COBS/ZPE should not be thought of as a general purpose compression algorithm, and it is not intended to compete with more sophisticated (and more computationally expensive) compression algorithms such as Huffman encoding [Huff52] [Knu85] and Lempel Ziv [LZ77] [Wel84]. Although, like PPP, these compression algorithms 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. It is also not clear that there is a great benefit to applying heavyweight compression at the link layer, since 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. In the case of data that has been encrypted, it is not possible to apply heavyweight compression at the link layer because data that has been properly encrypted is not compressible [Pra97]. ZPE is a lightweight technique that works well for small packets that contain zero pairs, without sacrificing the primary benefit of COBS: an aggressive bound on worst-case overhead, even for packets that may be large and contain no zero pairs.

Chapter Four

Theoretical Analysis

This chapter presents a theoretical analysis of the behaviour of COBS and COBS/ZPE in comparison to two other techniques, HDLC bit stuffing and PPP byte stuffing. It compares the best case, worst case, and average case encoding overhead, given uniformly distributed random data.

HDLC and PPP were chosen for comparison because both work reliably and are relatively efficient, and as a result both are widely used in today's software. They are arguably the canonical examples respectively of bit and byte stuffing.

It is useful to calculate the expected performance for uniform random data, because data that is properly compressed and/or encrypted has a uniform distribution of byte values. Data that is well-compressed must in general have a uniform distribution of byte values, because if it did not then Huffman encoding [Huff52] could be used trivially to compress the data further, contradicting the description of the data as being 'well compressed'. For similar reasons encryption algorithms seek to maximize the entropy of the data so as to maximize the strength of the encryption [Pra97]. Compressed data already makes up a large part of Internet traffic: For example, World-Wide Web traffic has grown to exceed all other Internet traffic sources combined, whether measured by number of packets, number of bytes, or number of flows [Braun94] [Thom97] and furthermore, 65% of World-Wide Web traffic is compressed GIF or JPEG image data [Gwert96]. I expect to see an increasing awareness of security issues on the Internet in the future, resulting in a similar increase in encrypted data. However, at the present time not all traffic is compressed and/or encrypted, so actual performance on today's real packets is shown in Chapter 5.

The chapter concludes with a discussion of the trade-off between minimizing the encoding overhead and minimizing the delay imposed on the stream of data being encoded.

4.1 Best Case Overhead

The best case for HDLC bit stuffing is a packet that contains no occurrences of five binary ones in a row. In this case, HDLC encoding adds no overhead to the packet at all.

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.

The best case for COBS/ZPE is a packet composed entirely of zeroes. In this case, each pair of zeroes is encoded as a one-byte code, 0xE1, resulting in a net compression effect (a 'negative overhead'). As with basic COBS, a phantom zero is still appended to the tail of every packet, so the best-case encoded size is half of the length n of the packet plus one byte, rounded up to an integer number of bytes:

$$\left\lceil \frac{n+1}{2} \right\rceil \text{ bytes}$$

This case of an all-zeroes packet gives a best case *saving* of:

$$n - \left\lceil \frac{n+1}{2} \right\rceil = \left\lfloor \frac{n-1}{2} \right\rfloor \text{ bytes}$$

4.2 Worst-Case Overhead

The worst case for HDLC bit stuffing is a packet that contains nothing but binary ones. In this case, HDLC encoding adds one bit for every five, giving an overhead of 20%.

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 both COBS and COBS/ZPE is a packet that contains no zeroes at all. In the case of basic COBS, 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. The maximum overhead is $^1/_{254}$ (roughly 0.4%) of the packet size, rounded up to a whole number of bytes. 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 6 bytes to a packet of this size. In the case of COBS/ZPE, each sequence of 223 bytes of packet data is encoded using 224 bytes of output data, giving one byte of overhead for every 223 bytes of packet data. The maximum overhead is $^1/_{223}$ (roughly 0.45%) of the packet size, rounded up to a whole number of bytes. For example, in the worst possible case COBS/ZPE would add an overhead of 7 bytes to a maximum size Ethernet packet.

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 over 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 consumed by 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, in general, over all packets, there must be some overhead. (Otherwise, if this were not the case, alphabet reduction could be applied repeatedly to compress any data indefinitely, without limit.)

To encode a packet of n bytes, we need to encode n symbols from an alphabet of 256 characters using some number of symbols, m, from a smaller alphabet of only 255 characters. We know that n symbols from an alphabet of 256 characters can encode 256^n different values and m symbols from an alphabet of 255 characters can encode 255^m different values, so we need to find m such that: $255^m \geq 256^n$

$$255^m \geq 256^n$$

$$m \log(255) \geq n \log(256)$$

$$m \geq n \frac{\log 256}{\log 255}$$

$$m \geq 1.000706319n$$

Thus, in general, over all possible inputs, any encoding of symbols from an alphabet of 256 characters using symbols from an alphabet of 255 characters must increase the number of symbols by a ratio of at least:

$$\frac{\log 256}{\log 255}$$

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 whose average performance over all packets can beat this bound. Perhaps more significantly, this bound on average performance tells us that no algorithm can ever have a worst-case bound better than 0.07063%. If any algorithm did have a worst-case bound better than this, that would necessarily mean that its average performance over all inputs would also be better than 0.07063%, and we know that cannot be the case.

In practice we can take advantage of the fact that our network traffic is not purely random data, by recognising that certain patterns of data occur more frequently than others and devising an algorithm that encodes this particular mix of data more efficiently. This allows us to improve our average performance for typical network traffic, but it does not change the fact that the worst case can never be improved beyond the theoretical limit.

4.3.1 Expected Overhead for HDLC

In [Pap87] Papastavridis shows that the mean number of stuffing bits in a prefix-synchronized code such as HDLC is approximately:

$$\frac{n}{2^k - 1}$$

where the data is n bits long, the prefix is k+1 bits long, and n»k. For HDLC, the prefix referred to is the sequence 0111111, so k=6. Thus Papastavridis shows that the mean number of stuffing bits in HDLC encoded data is roughly 1 bit in 63. Note that Papastavridis shows his result in terms of the proportion of stuffing bits expected to be *observed in the final output data*, not in terms of the number of stuffing bits the encoding algorithm is expected to *add to the input data*. If in the output 1 bit in 63 is found to be a stuffing bit, that is equivalent to saying that the encoding algorithm added one bit for every 62 bits of input data. Hence, in terms consistent with those used throughout the rest of this dissertation, HDLC bit stuffing adds an average overhead of 1 bit in 62, or a little over 1.6%.

One way to understand HDLC's average overhead intuitively is as follows: Given a long sequence of bits that has not yet generated a stuffing event, what is the probability that the next bit will generate a stuffing event? The necessary and sufficient condition for the next bit to generate a stuffing event is that the last five bits of the sequence are currently 01111 and the new bit added is a 1, resulting in a new sequence that now ends with 011111. Only when a sequence ends with this pattern does a stuffing event occur. This particular six-bit pattern that we are interested in is only one of many possible patterns that could terminate a sequence. A sequence can end with any of the 64 possible combinations of six bits, *except* for 111110 and 111111. A sequence cannot end with either of these patterns, because we are calculating probability *given that the sequence has not yet generated a stuffing event*, and if the sequence ended with either of these two patterns we know it would have already generated a stuffing event previously. Thus only one six-bit pattern out of the 62 allowable six-bit patterns generates a stuffing event, meaning that each bit has a $^1/_{62}$ chance that it will cause a stuffing event.

This simple analysis ignores boundary effects near the start of a sequence, such as the fact that the first four bits of a sequence can never generate a stuffing event, and the fifth bit has a disproportionately high $^1/_{32}$ probability of causing a stuffing event, but nevertheless it is useful to see that the result produced by this simplified analysis of the problem supports Papastavridis's far more rigorous (and perhaps less intuitive) solution.

4.3.2 Expected Overhead for PPP

For PPP the expected overhead is easy to calculate. PPP has only two distinct behaviour 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 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, giving 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.

4.3.3 Expected Overhead for COBS

For COBS the average overhead is a little harder to calculate than for PPP, since COBS has 255 different behaviours, rather than just the two that PPP has. In addition, not only does each behaviour write a different number of output bytes, each behaviour 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(0x01) 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(0x02) 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:

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

The longest code block, code 0xFF, occurs when the algorithm encounters 254 non-zeroes without seeing a single zero. The probability p(0xFF) 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:

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

Now we calculate the average output per code block. The probabilities of each different behaviour 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:

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

The ratio of average output divided by average input is:

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

The theoretical average overhead for COBS on random data is therefore a little under 0.23%. This is about $3^1/_4$ times worse than the theoretical optimum, more than three times better than PPP's average.

4.3.4 Expected Overhead for COBS/ZPE

As with basic COBS, we calculate the probability of each code block, and the number of bytes of input and output for each code block. The probability p(0x01) that the first byte is a zero and the second is not, resulting in a code 0x01 block, is $^1/_{256} \times {}^{255}/_{256}$. The probability p(0xE1) that both the first and the second bytes are zero, resulting in a code 0xE1 block, is $^1/_{256} \times {}^1/_{256}$. Table 31 enumerates, for each code of the form *k*+*n*, the probability of code block *k*+*n* occurring, and the number of input and output bytes for that code block.

The ratio of average output divided by average input is:

$$\frac{\sum_{n=1}^{255} \mathrm{p}(n) \times \mathrm{out}(n)}{\sum_{n=1}^{255} \mathrm{p}(n) \times \mathrm{in}(n)} \approx 1.002800$$

Base Code k	Offset n	Probability $p(k+n)$	Input $in(k+n)$	Output $out(k+n)$
$k = 0x00$	$1 \le n < 0x20$	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256} \times \frac{255}{256}$	n	n
$k = 0x00$	$0x20 \le n < 0xE0$	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256}$	n	n
$k = 0x00$	$n = 0xE0$	$\left(\frac{255}{256}\right)^{n-1}$	$n - 1$	n
$k = 0xE0$	$1 \le n < 0x20$	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256} \times \frac{1}{256}$	$n + 1$	n

Table 31. Code Block Probabilities for Consistent Overhead Byte Stuffing with Zero-Pair Elimination

The theoretical average overhead for COBS/ZPE on random data is therefore about 0.28%, which is consistent with what we would intuitively expect: Compared to basic COBS, COBS/ZPE reduces the maximum code block length by about 12%, so we would expect the average code block length also to reduce by about 12%. Also, because the maximum code block length is reduced by about 12%, we would expect that the probability that any particular byte sequence reaches that maximum length, thereby incurring a byte of overhead, would increase by about 12%. Hence, the average code block is 12% shorter, and the probability that it will incur a byte of overhead is 12% higher, which would suggest that the overall average overhead would go up by 24%. This approximate intuitive result is consistent with the result given by the numerical calculation, that the overhead increases from about 0.23% to 0.28%.

4.4 Encoding Delay

One possible criticism of COBS encoding is that it appears to add delay to data transmission, whereas HDLC and PPP do not. The argument is that as each bit is fed to an HDLC encoder, the encoder immediately generates its corresponding output, namely that bit, or that bit followed by a zero stuffing bit, as appropriate. Likewise, as each byte is fed to a PPP encoder, the encoder immediately generates its corresponding output, namely that byte, or a pair of bytes, as appropriate. In both cases, for each unit of data the output generation is immediate. The encoder does not have to wait until it has seen some amount of subsequent input before it is able to decide what its output will be. In contrast, a COBS encoder may have to buffer up to 254 bytes of data internally before it generates any output, which suggests that COBS encoding could add a significant amount of delay to the data transmission pipeline.

This section addresses this possible criticism in two ways. The first is to show that delay and worst-case overhead are intrinsically linked, so any delay in the COBS algorithm is not so much a flaw peculiar to the COBS algorithm, but a necessary consequence of achieving a low bound on worst-case overhead. The second is that the apparent dependency on future data, and the consequent apparent encoding delay, are due to considering bytes as the basic unit of data. If we consider the network packet as the basic unit of data, then there is no dependency on future data, and thus no encoding delay.

4.4.1 Relationship Between Delay and Worst-Case Overhead

Any data stuffing algorithm is effectively a finite state machine that reads input, undergoes state changes, and writes output. Since in the worst case the output is necessarily longer than the input, for each n units of data read some number $m>n$ units of data may have to be written. To minimize worst-case overhead, we should make m as small as possible. Unless we plan to allow writing of fractional units of data, which is beyond the scope of this dissertation, m must be an integer, and the smallest integer greater than n is $n+1$. Thus the worst-case encoding ratio can be no better than ${}^{n+1}/_{n}$. Consequently, to minimize worst-case overhead, we should make n as large as possible. The smaller the value of n, the larger the worst-case encoding overhead. For PPP the amount of data read before an output is generated is always one byte, so $n=1$, and the worst-case encoding ratio is 2:1. For COBS the amount of data read before an output is generated may be up to 254 bytes, so $n=254$, and the worst-case encoding ratio is 255:254.

Upon casual inspection it may appear that HDLC is able to beat this performance limit, since its worst-case overhead is only 20%, yet it is always able to generate an output for every single bit of input. However, this interpretation is misleading, because HDLC encoding transfers the delay to the receiving end of the pipeline. While the HDLC encoder is able to process individual bits with no delay, the HDLC decoder is not. When the HDLC decoder observes a series of one-bits, it is not able to say whether those one-bits are data bits or the end-of-frame marker until it has examined the sixth bit to see whether it is a zero or a one. Hence in the case where it receives five ones followed by a zero, the HDLC decoder has to read six bits before it can generate any output, at which time it can then immediately generate five bits of output. Thus HDLC has a worst-case encoding ratio of 6:5, and at the receiving end there is a decoding delay of up to five bits before the meaning of a particular bit can be properly decoded.

To have low overhead we need to have a large value of n, and to have a large value of n we need to add delay, either at the transmitting side or at the receiving side, or a combination of the two. This suggests the unhappy conclusion that it is impossible to have a small worst-case overhead without also adding a large amount of delay but, as shown below, this apparent problem may not matter at all.

4.4.2 Units of Data for Encoding Algorithms

At the beginning of this section, I stated that both HDLC and PPP encoders generate immediate output for each unit of input they are given. However, for HDLC the unit of input was a single bit, while for PPP the unit of input was the byte. If we consider the input to the PPP encoder to be a stream of bits instead of bytes, then its output is no longer always immediate. Upon being presented with the first bit, the PPP encoder may be unable to generate any output. It may have to wait until it has received the next seven bits to complete the entire byte before it knows what output it should generate. However, in practice, PPP encoders are not presented with data one bit at a time. They are presented with data at least a byte at a time. In fact, in most networking software, the PPP encoder is not even presented with single bytes; it is presented with an entire packet to encode, and the entire packet is encoded in a single *while* loop. One of the reasons for processing data a packet at a time is that popular network protocols such as Ethernet [Tan88] [IEEE802.3] and IP [RFC791] do not carry data as an unstructured stream of bits or bytes over virtual (or physical) circuits, but as packets, and packets are the atomic units of data delivery. Sending or receiving a partial packet is rarely a useful operation.

In the same way that PPP encoding is more usefully applied at a per-byte level than at a per-bit level, COBS encoding is more usefully applied at a per-packet level than at a per-byte level. Although the COBS encoding of a given byte in general depends on the values of other nearby bytes, the encoding of any given packet is idempotent, and is not dependent on knowledge of any previous or subsequent packets. Since the kinds of networks considered in this dissertation use packets as their basic units, it is reasonable to relax the restriction that the byte stuffing process must be an idempotent per-byte operation, and this ability to consider more information at once is what allows COBS to achieve a much better worst-case bound than HDLC or PPP.

4.5 Summary

Figure 8 shows a comparison of the results for PPP and for COBS, to illustrate visually the dramatic difference between the narrow range of overheads generated by COBS and the wide range of overheads that can be generated by PPP.

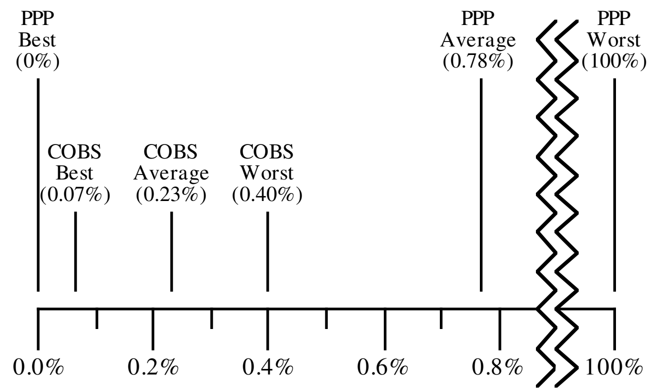


Figure 8. Encoding Overhead for 1500 Byte Packet

> 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.

These average 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 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, I encoded traces of real-world network traffic using both COBS and PPP byte stuffing, and these results are presented in the next chapter.

Chapter Five

Experimental Results

This chapter presents the encoding behaviour of COBS and other protocols for real-world network traffic. Real-world network traffic may not behave the same way as Chapter 4's 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, I gathered traces of network traffic using tcpdump [Jac89] and compared how efficiently those packets were encoded by HDLC, PPP, COBS and COBS/ZPE.

I captured traces of traffic sent both over Metricom packet radio interfaces [Che96] and over Ethernet [Tan88] [IEEE802.3]. The reason for studying packet radio traffic as well as Ethernet traffic was because unlicensed FCC Part 15 radio devices were one of the main motivations behind the development of consistent overhead byte stuffing. The performance of wide-area radio interfaces is dramatically worse than that of an Ethernet (both in bandwidth and in latency) and this performance difference may have an influence on the way people use their network connection. Consequently, I wanted also to study traffic patterns in that environment as well as in the more common Ethernet environment.

Two traces are presented here. The first is a wireless traffic trace, consisting of predominantly small packets. Just over half of the packets (51%) in this trace are either TCP ack packets containing no data, or single-character TCP data packets, such as a single keystroke on a remote login connection. The second trace is a large file transfer containing roughly two thirds maximum-sized TCP data packets and one third TCP acks. (63.6% of the IP packets were maximum-sized TCP data packets and 36.3% were TCP acks.)

For each trace, the packets were encoded using HDLC, PPP, COBS and COBS/ZPE. For each method a histogram shows overheads in the range ±30 bytes on the horizontal axis, and for each of those overheads, the percentage of packets incurring that overhead. This percentage is plotted on a log scale to show more detail at the low end. For these traces 0.001% of the packets amounts to less than one packet, and since the only number of packets less than one is zero packets, 0.001% is chosen as the base line of the log scale.

In all of our traces, PPP's worst-case encoding was always significantly higher than COBS's worst case. Even so, PPP never came close to using the full factor-of-two overhead that is possible. However, the fact that we never observed these pathological packets does not mean that it is safe to engineer a PPP implementation that does not take them into account. There is at least one example of real-world traffic that exhibits frequent reproducible pathological PPP encoding behaviour: voice over IP using G.721 [G721]. During periods of near-silence, this encoding can sometimes transmit data that is almost entirely 0x7E byte values, causing pathological PPP data expansion [Carl97].

5.1 Three-Day Trace

One of my colleagues frequently works at home, and his sole Internet connection is via a portable ISM-band packet radio attached to his laptop computer. I collected a trace of all his packets for a period of three days. The goal was to capture a representative trace of packets from 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 the wireless interface in this case was 1024 bytes, giving a worst-case COBS overhead for large packets of 5 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.

The three-day trace is 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 9.

HDLC incurred a total overhead of 703,607 bits (0.87%). Over 75% of the packets incurred from one to ten bits of overhead, but a few packets incurred over three hundred bits of overhead.

PPP incurred a total overhead of 36,069 bytes (0.36%). 74% of the packets incurred no overhead, but some packets incurred a significant amount. More than 100 packets incurred fifteen bytes of overhead or more, and one packet fell beyond the scale of the graph with an overhead of fifty-three bytes. In this trace no packets incurred more than fifty-three bytes of overhead, supporting the belief that most real packets do not come close to the factor-of-two overhead that conventional byte stuffing forces us to design for.

For COBS the overhead is concentrated in a tight spike in the middle: every packet incurred one to four bytes of overhead. 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 might 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 I captured a large bulk transfer of compressed image data. The data file was MIRV.MPG, a 15.3MB MPEG [ISO11172] 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 5 bytes.

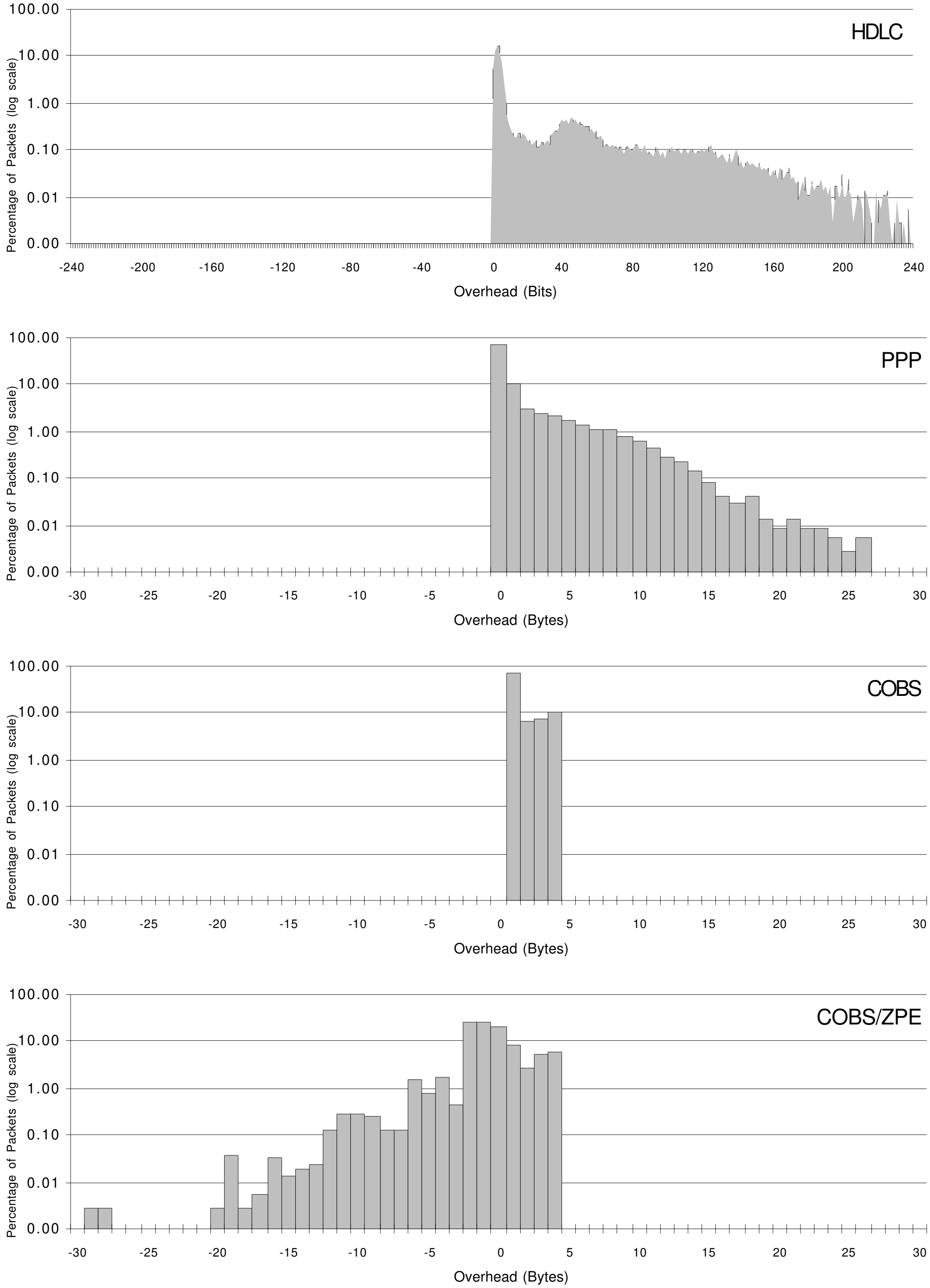


Figure 9. Encoding Overhead Distribution for 3-Day Trace

Histograms showing, for each amount of overhead indicated on horizontal axis, the percentage of packets that incur that overhead. All histograms are drawn to the same scale and labelled in bits or bytes as appropriate.

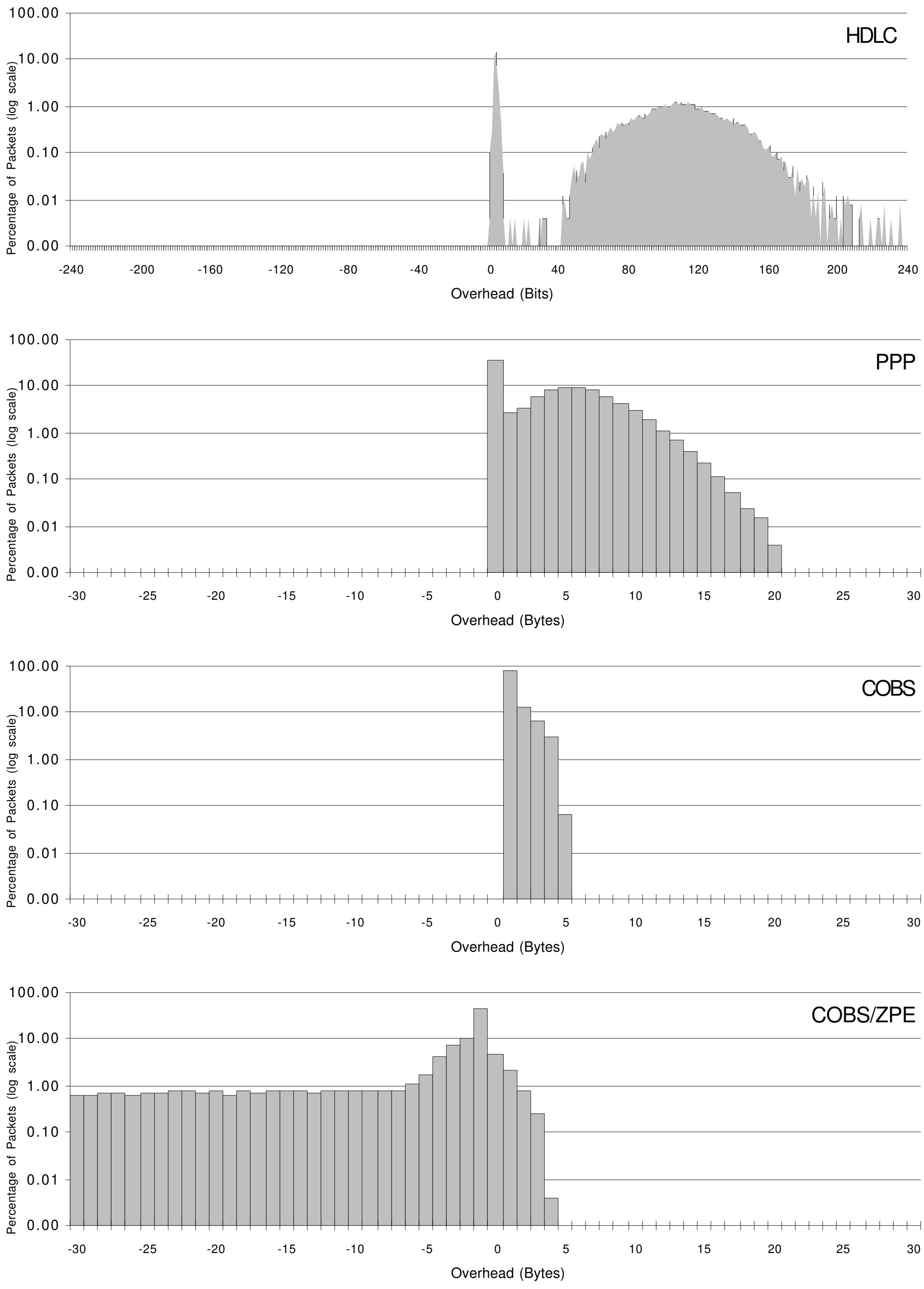


Figure 10. Encoding Overhead Distribution for MPEG Trace

Histograms showing, for each amount of overhead indicated on horizontal axis, the percentage of packets that incur that overhead. All histograms are drawn to the same scale and labelled in bits or bytes as appropriate.

The MPEG 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 10.

HDLC incurred a total overhead of 1,862,796 bits (1.27%) and this histogram has two visibly separate peaks. The first peak, one to eight bits, is the result of the overhead from short (40-byte) ack packets. The second peak, forty to two hundred bits, is the result of the overhead from the long (1088-byte) data packets.

PPP incurred a total overhead of 101,024 bytes (0.55%). 36% of the packets (mostly the ack packets) incurred no overhead. The majority of packets incurred one to ten bytes of overhead and one packet incurred as much as twenty bytes of overhead. In this trace no packet incurred more than twenty bytes of overhead, supporting the belief that most 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%), and the overhead is concentrated in a tight spike in the middle: every packet incurred between one and five bytes of overhead. 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%.

Chapter Six

Extending COBS

This chapter describes four ways of extending the basic COBS concept: (i) using COBS to eliminate a data value other than zero, (ii) applying the COBS concept to other word lengths in addition to eight-bit bytes, and (iii) extending the COBS concept to eliminate multiple values from the packet data, and (iv) using COBS bytes-to-packets framing to also eliminate the overhead of lower-level bits-to-bytes framing,

6.1 Eliminating a Value Other Than Zero

As described, COBS first searches the packet for all occurrences of zeroes (the *search value*), logically dividing the packet into one or more search-value-terminated chunks, and then encodes those chunks to produce an output stream that is free of zeroes (the *eliminated value*). However, the framing value zero may not be appropriate in all situations. For example, a radio interface that connects to the computer through a serial port like a Hayes modem might use an ASCII carriage return (byte value 0x0D) to mark the end of each packet [Che96]. There are two ways to modify COBS to eliminate a different value. The first is to change the search value, which inherently results in a different value being eliminated. The second is to keep zero as the search value, and use an output mapping function to change just the eliminated value.

6.1.1 COBS with a non-zero Search Value

Using some value s other than zero as the search value simply requires a redefinition of the meanings of the code bytes. The implicit trailing byte on the end of each code block is defined to be a byte with value s instead of a zero byte, and value s is defined to be illegal and never used as a code byte. This transformation of the algorithm results in output that contains no occurrences of the value s.

In this s-eliminating variant of COBS, the special maximum length block (254 data bytes without any implicit trailing byte) is indicated by code value m, where m=0xFF if $s \neq$0xFF, and m=0xFE otherwise. Code value x ($0 \leq x < s$) indicates a block of x bytes with an implicit trailing byte with value s and code value x ($s < x < m$) indicates a block of $x-1$ bytes with an implicit trailing byte with value s.

There are three minor factors that influence the choice of s:

The first factor is that many CPUs, especially simple eight-bit CPUs, are able to test a register for equality with the value zero more efficiently than they are able to test for equality with some other arbitrary value. This favours the use of zero as the search value.

The second factor is that when zero-terminated chunks are used, all the valid codes fall in a contiguous range: code byte zero is illegal, code byte 0xFF is a special exception, and each of the remaining code bytes n in the range from 0x01 to 0xFE encodes a block of exactly n data bytes. This relationship helps simplify implementation. If some byte other than 0x00 or 0xFF is chosen as the illegal value, then the range of valid code bytes is not contiguous, and the above relationship that code n encodes a block of n bytes is not true. The valid code bytes are split into two disjoint ranges, and the decoding algorithm has to handle codes in the lower range differently to codes in the upper range.

The third factor is that COBS encodes short chunks with no overhead. COBS encoding only adds overhead when a chunk is longer than 254 bytes and one or more special maximum-length code blocks has to be used to encode that chunk. Consequently, if we can pick a search value that is found generously distributed throughout our packet data, most or all of the chunks will be short, and less overhead will be added. Because the value zero is common in binary computer data, using the value zero may help reduce the number of oversized chunks in some cases. However, it should be noted that this is an extremely minor effect. The best case for COBS encoding is one byte per packet, and the worst case is one byte for every 254 bytes of packet data (0.4%), so any variation in overhead resulting from wise or poor choice of search value can only fall within this narrow range.

In the cases where these factors apply, they favour implementing the standard zero-eliminating COBS algorithm.

In cases where the value required to be eliminated is not the same as the preferred search value, this can be achieved by using output mapping instead, as described below.

6.1.2 COBS with Output Mapping

Standard COBS output contains no instances of value zero. There are a number of ways that this zero-free output can be easily converted to eliminate some other byte value *e* instead, if that is required.

One way is simply to replace all occurrences of *e* in the data with zero. Upon reception, the receiver simply replaces occurrences of zero in the data with value *e* before decoding. Since we know that the original COBS output had no zeroes this is an unambiguous transformation, because we know that any appearances of zero in the data can only have occurred as a result of replacing occurrences of *e*.

Conditional code of this kind (‘if (x==0) x=e;’) can be inefficient on some processors because branches cause bubbles in the execution pipeline. As an alternative to selective replacement, all bytes can be XORed with *e* before transmission. The only value that gives the result *e* when XORed with *e* is zero, and since there are no zeroes in the output from COBS, this procedure cannot result in any occurrences of *e* in the converted output. The receiver XORs each received byte with *e* to reverse the transformation before feeding it to the COBS decoder, so the data is decoded correctly.

Any similar transformation, such as using binary eight-bit add and subtract, could also be used equally well.

Finally, a 256-entry lookup table can also be used to implement any desired eight-bit to eight-bit mapping function.

Although the transformations are described here as a post-processing step after encoding, in real implementations the COBS encoding and the output conversion are performed in a single loop for efficiency reasons [Cla90] [Ahl96a] [Ahl96b] [Ahl96c] [Braun96].

The remainder of this dissertation will consider only the case of eliminating zeroes from the data. Any of the processes described above may be used in any case where a different value is to be eliminated.

6.2 Using Other Word Lengths

This dissertation describes COBS using bytes as the unit of data, but the same concept applies equally well to any word length $w>1$. For word length w, value zero is not used as a code value, value 2^w-1 is used to indicate a run of 2^w-2 non-zero values, and all the remaining values n are used to indicate a sequence of $n-1$ non-zero values followed by an implicit zero. Table 32 summarizes these code values.

Using Consistent Overhead Word Stuffing (COWS) with word length w, the best case is one word of overhead per packet, and the worst case is one word for every 2^w-2 words of packet data. In both cases there is the overhead of the framing marker to consider, which is w bits for a word-oriented interface, $2w$ bits for a serial interface using synchronous idle markers, and $2w+1$ bits for a serial interface using asynchronous idle markers.

Code	Followed by	Meaning
0	(not applicable)	(not allowed)
n	($n-1$) data values	The ($n-1$) data values, followed by a single zero
2^w-1	2^w-2 data values	The 2^w-2 data values, **not** followed by a zero

Table 32. Code Values used by Consistent Overhead Word Stuffing

> The concept of Consistent Overhead Byte Stuffing can be applied to any word length $w>1$.

As the word length increases, the size of the fixed overhead (one word plus the framing marker) increases, but the variable component of overhead, consequently the worst-case bound, decreases. Table 33 shows the total overheads (encoding overhead plus framing marker overhead) for sending a maximum-sized 1500-byte Ethernet packet using synchronous idle markers, for word lengths of 4, 8 and 16 bits.

When using sixteen-bit words, the best-case and worst-case are identical for a 1500-byte packet. This is because COWS with a sixteen-bit word is able to encode any sequence less than 65534 words long (131,068 bytes) using a single code block. Variable overhead could only be incurred for packets larger than 131,068 bytes.

Word length	Best-case overhead	Worst-case overhead
4	12 bits (0.1%)	868 bits (7.2%)
8	24 bits (0.2%)	64 bits (0.5%)
16	48 bits (0.4%)	48 bits (0.4%)

Table 33. Overhead Values for Consistent Overhead Word Stuffing

Total overheads for sending a maximum-sized 1500-byte Ethernet packet using synchronous idle markers, for word lengths of 4, 8 and 16 bits.

For packets of around 1500 bytes, COBS with eight-bit words achieves a good worst-case bound while adding only a small fixed overhead. Using four-bit words halves the fixed overhead but makes the worst-case bound fourteen times worse. Using sixteen-bit words only slightly reduces the worst-case bound, but it doubles the fixed overhead, which adversely affects all packets, even those much smaller than 1500 bytes. For this reason, this dissertation concentrates on COBS using eight-bit words, not only because it is convenient to implement in software, but also because it is the most appropriate word size for COBS encoding when used to encode a typical Ethernet-like range of packet sizes.

6.3 Eliminating Multiple Values

This dissertation describes COBS used to eliminate a single eight-bit value from packet data, to free that value for use as a framing marker. There are some situations where it may be desirable to eliminate more than one value from the data, such as when sending data over some non-transparent lower layer that forbids certain values, and the COBS (or COWS) concept can also be extended to handle this.

Using a word length w, there are 2^w possible values. If we wish to eliminate k values from the data, then that leaves $2^w–k$ values available for use as COWS code and data values. Reserving one code value (code $2^w–1$) for the special maximum length code, that leaves $2^w–(k+1)$ remaining COWS code values to be assigned. The first available code is assigned to represent the first value to be eliminated. The next code is assigned to represent the second value to be eliminated, and so on until the first k codes have been assigned. The next k codes are assigned to represent a single data byte followed by an implicit value. The next k codes are assigned to represent a pair of data bytes followed by an implicit value, and so on until there are fewer than k codes remaining.

Using COWS with word size w to eliminate k values, the best case encoding overhead is one word per packet, and the worst case is one word for every

$$\left\lfloor \frac{2^w - 1}{k} \right\rfloor - 1 \text{ words of packet data.}$$

The maximum number of values that a COWS-type algorithm can eliminate is one fewer than half of the values in an alphabet. This is because each of the eliminated values must have at least one code value that implicitly represents it, or there would be no way to represent that particular value. Since there are $2^w-(k+1)$ COWS code values available to be assigned, that means $k \le 2^w-(k+1)$, so $k < 2^{w-1}$.

Table 34 below illustrates an example assignment of codes to make an encoding that simultaneously eliminates two different values from eight-bit packet data.

Code	Followed by	Meaning
0, 1	(not applicable)	(not allowed)
$2 \le n < 254$	$\lfloor \frac{n}{2} \rfloor - 1$ data values	The $\lfloor \frac{n}{2} \rfloor - 1$ data values, followed by an implicit zero if code is even, or an implicit one if code is odd
254	(not applicable)	(unused code)
255	126 data values	The 126 data values, **not** followed by any implicit values

Table 34. COBS Code Values used to Eliminate a Pair of Byte Values

> The concept of Consistent Overhead Byte Stuffing can be extended to eliminate more than one value from the data, such as when sending data over a non-transparent lower layer that forbids certain byte values.

Using the codes defined in Table 34, the number of data bytes for code value $n<255$ is given by the integer quotient of $^n/_k$, and the implicit value to be appended is given by the remainder of $^n/_k$. However, decoding packet data can be done without requiring any division operation, simply by initializing a counter to $n-k$ when the code byte is read, and decrementing the counter in steps of k. When the counter reaches a value less than k all of the data bytes in that code block have been read, and if the initial value was other than 255, then the value remaining in the counter is the implicit byte value to be appended.

While it is possible to extend COBS in this way, and it may be useful to do so for certain applications, the aggressive worst-case bound which makes COBS useful is rapidly weakened as the number of values to be eliminated increases. For example, using COBS to eliminate 34 different byte values, as is sometimes done for PPP links over non-transparent networks, results in a worst-case overhead of one byte for every six bytes of packet data ($16^2/_3$%).

For this reason, this dissertation concentrates on using COBS to eliminate a single eight-bit value in order to facilitate reliable unambiguous packet framing, not extensive byte stuffing to compensate for non-transparency of the underlying network.

6.4 Lower-Level Framing

COBS is defined in terms of byte operations. This means that there also needs to be some underlying mechanism to detect where the byte boundaries fall in the bit-stream. When used over an RS-232 serial port, the start/stop bits perform this function (not very reliably), at a cost of 20% extra overhead. HDLC framing is usually more efficient, 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, with the choice of an appropriate bit-level framing pattern, COBS-encoded data becomes self-framing, and no lower-layer bit stuffing is needed. All that is required to make this work is a bit-level framing pattern chosen so that it can never occur anywhere in COBS-encoded data. If the framing pattern cannot inadvertently appear, there is no need for any bit stuffing mechanism to eliminate it.

Finding an appropriate 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 a candidate for the COBS end-of-packet marker: a run of fifteen (or more) zero-bits.

The work of Barker [Bar53] and Artom [Art72] gives further guidance about the choice of an appropriate framing pattern. The framing pattern should not only be one that does not appear in the data stream, but also one that cannot appear in any overlap between the marker and the data stream. A framing marker consisting of just a run of zero-bits would not meet this requirement. It could never appear entirely within the encoded packet data, but appending this framing marker to the end of an encoded packet could result in an ambiguous situation. This is because nothing prevents an encoded packet from ending with as many as seven consecutive zero-bits. Upon seeing a long run of zero-bits, the receiver does not have any simple reliable way to tell which, if any, of those zero-bits actually belong to the tail of the previous packet, and which bits belong to the framing marker. This ambiguity can be solved by refining the definition of the framing marker to be a one-bit followed by fifteen zero-bits:

```
1000000000000000
```

Unintended occurrences of the framing marker are now eliminated, because any sequence made by concatenating bits from the end of a packet with bits from the start of the marker will result in a sequence that has at least one one-bit somewhere in the middle, and this cannot be mistaken for a framing marker because the framing marker has no one-bits in the middle.

COBS bit-level framing can be defined to use a synchronous idle sequence, or an asynchronous idle sequence, as appropriate to the application requirements. Using a synchronous idle sequence, the end-of-packet marker is a one-bit followed by exactly fifteen zero-bits. If the link is idle and there are no packets to transmit, then these 16-bit end-of-packet markers are transmitted back-to-back continuously until there is a packet to transmit. Using an asynchronous idle sequence, the end-of-packet marker is a one-bit followed by fifteen *or more* zero-bits. In this case, after the basic end-of-packet marker has been sent, an idle line can simply be filled with as many zero-bits as are needed to bridge the gap until there is another packet to send. So that the receiver knows when the next packet begins, the asynchronous framing marker is defined to be a one-bit, followed by fifteen or more zero-bits, and then another one-bit, as shown below:

```
100000000000000 ... 01
```

For a line that is mostly busy the synchronous idle specification is marginally better, because the framing marker is only 16 bits long, compared to the 17-bit minimum length of the asynchronous idle framing marker, so there is slightly less overhead.

For a line that is mostly idle, the asynchronous idle specification is marginally better, because packets can be sent with slightly less delay. Using synchronous idle markers, if an idle line has just begun transmitting its next idle end-of-packet marker when an outgoing packet arrives at the interface, the packet will have to wait for 15 bit-times before it can be sent. Using asynchronous idle, the interface could immediately send the start bit and begin sending the packet after a delay of only one bit-time.

Used with either of these framing marker definitions, 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.

Chapter Seven

Applications

There are many applications, both current and future, that could benefit from consistent overhead byte stuffing. This chapter presents some major ones, and explains how consistent overhead byte stuffing either improves on an existing solution, or how it makes a new kind of solution possible.

7.1 Conventional Serial Line Byte Stuffing

COBS would be a useful replacement for conventional two-for-one byte stuffing algorithms even when used with modems over conventional telephone lines. Fortunately PPP has an option negotiation mechanism that allows new extensions to be added to the PPP protocol without sacrificing backward compatibility with older implementations. The IETF PPP Working Group is currently considering an Internet-Draft specifying just such an extension which allows PPP implementations to negotiate use of a new framing scheme based on an extension of COBS/ZPE [Carl97].

7.2 Slotted Media Access Protocols

Some media access protocols, such as Slotted Aloha [Abr72] [Abr85] use fixed sized transmission slots. In order to ensure that no packet can exceed its allocated slot size, the sender must artificially limit its maximum payload size to account for worst-case expansion. In the case of a system using conventional two-for-one byte stuffing, this would mean that no packet's raw size could be more than half of the slot size, effectively wasting half of the channel's bandwidth.

Consistent overhead byte stuffing solves this problem by providing much better predictability in the encoded size of the data. Because we know no packet can increase by more than 0.4%, we can set the maximum payload size to 99.4% of the slot size, achieving almost double the utilization of a conventional two-for-one byte system.

7.3 Simplified MACA Protocol

The MACA protocol [Karn90] provides another example application of COBS. In this protocol, after a successful RTS/CTS exchange (Request To Send/Clear To Send), other stations must remain quiet for long enough for the requesting station to transmit its packet, so that they do not interfere with it. In an environment where PPP byte stuffing is used to encode the packet, to be certain that the transmission is complete the stations might have to remain quiet for twice as long. With HDLC bit stuffing the uncertainty would not be so great, but the stations would still have to allow a 20% safety margin. (The stations that are required to remain quiet to prevent interference at the *receiver* are not necessarily in range of the *sender*. They may not be able to hear the packet transmission in order to detect when the transmission is complete and determine for themselves when it is safe to send again.) The sender could pre-compute the encoded size and send that information, instead of (or as well as) the raw payload size, in its initial RTS packet, but this is extra software overhead that could be avoided by using COBS. If the radio is using HDLC bit stuffing hardware to transmit the packet, then the software must pre-compute the HDLC overhead in order to put this information into the RTS packet and, as discussed before, computing bitwise encodings like HDLC in software is often not very efficient.

Consistent overhead byte stuffing solves this problem by providing much better predictability in the encoded size of the data. It eliminates the need to precompute the encoded size because we can instead simply use the raw size plus 0.4%, since we know the encoded size can never exceed this.

7.4 ISM Packet Radio Byte Stuffing

Packet radio devices, such as Metricom's 900Mhz packet radio devices [Che96] are constrained by United States FCC regulations to transmit for no more than 400ms at a time. This legally imposed limit presents a constraint it would be very unwise to break.

Using conventional byte stuffing which can potentially double packet size, the maximum payload size must be set to half the maximum transmission size. Using HDLC framing, the maximum payload size must be set to no more than 80% of the maximum transmission size. Using consistent overhead byte stuffing the maximum payload size may be set as high as 99.6% of the maximum transmission size.

7.5 Improves ATM Efficiency

On average, ATM Adaptation Layer 5 (AAL5) [RFC1483] wastes half an ATM cell — 24 bytes — at the end of every packet that is sent. Certain particular common kinds of packet suffer especially badly. For example, a single character telnet packet is typically 41 bytes long — 20 bytes of IP header, 20 bytes of TCP header, and one byte of data. When the 8-byte AAL5 CPCS-PDU trailer is added, this makes 49 bytes, so it just marginally fails to fit into a single 48-byte ATM cell and requires two cells instead. 96 ATM payload bytes are required to transport a 41-byte IP packet — an overhead of 134%. Even for larger packets, on the order of 1000 bytes long, an average wastage of half a cell per packet results in an overhead rate of 2.4% (not counting the ATM cell headers themselves).

By streaming IP packets back to back using COBS, that wastage could be reduced to less than 0.4% in the worst case, and 0.23% on average, plus ATM cell headers. Particularly for small IP packets where the AAL5 overhead is great, this is a big improvement.

It should be noted that streaming packets back to back this way does not preclude use of techniques such as Romanow-Floyd Early Packet Discard [RoFlo94]. Early Packet Discard is a technique where, if the ATM switch drops one single ATM cell, then all the subsequent cells belonging to that IP packet are also dropped, to avoid wasting link bandwidth by sending cells that will be useless to the receiver. When using COBS back-to-back streaming, the EOP (End of Packet) bit in the ATM cell header can still be used to indicate to the ATM switch where IP packet boundaries fall in the data stream. When an ATM switch is in discard mode, seeing a cell with the EOP bit set would indicate a cell containing the end of one packet (and in most cases the beginning of the next) and hence would serve to indicate to the switch that cell forwarding should be turned back on immediately, beginning with that cell.

7.6 File Storage

Sometimes large numbers of records are stored in a single storage medium. When this is done, some mechanism is usually desired to extract individual records from the file. Three examples are MPEG video disks, Unix Maildrop files, and backup tapes.

7.6.1 MPEG Video Disks

When MPEG [ISO11172] video is stored on CD-ROM there are typically as many as 100,000 individual records — video frames —concatenated together with little or no indexing information. If random access to a portion of the disk required reading the entire MPEG file from the beginning, it would take several minutes and the user experience would be extremely poor. If any part of the disk were damaged, then all data following that damaged area might be unrecoverable. In the absence of any indexing information it is desirable to be able to seek to any point on the disk and find where the next frame starts in an *ad hoc* fashion, using only the data that is there.

MPEG allows random access by beginning each frame with three zero bytes, so that the receiver can simply search for three zero bytes to find the next frame boundary. MPEG does not perform any kind of data stuffing to eliminate occurrences of zero triplets from the data. Instead, the MPEG committee defined the MPEG video encoding very carefully so that under no circumstances can three zero bytes ever appear anywhere in the body of an encoded video frame. This non-transparency of the underlying MPEG layer forced the specification of the MPEG video encoding standard to be considerably more complicated than it might otherwise have been. Unfortunately the MPEG audio encoding standard fails to ensure that no run of three zero bytes ever appears in encoded audio data, which means that searching for three zeroes is actually not a reliable way of finding the next frame boundary.

Using COBS framing, the video encoding specification could be a lot simpler, and frame boundaries could be accurately located by simply searching for the framing marker.

7.6.2 Unix Maildrop Files

The Unix electronic mail software ‘sendmail’ concatenates received mail messages together into a single file. The start of each new message is detected by searching for a line beginning with the word ‘From’. If this word appears anywhere in the body of the message, then a kind of 'word stuffing' is performed, replacing 'From' with '>From' before the message is added to the combined mail file. This transformation is not reversed when the mail is delivered, resulting in some mail messages becoming changed in transit. When a human being is reading the message, the addition of extra '>' characters at the start of lines may be slightly confusing and unsightly, but not catastrophic. However, if the text of the message is, say, Base-64 encoded binary data that by misfortune simply happens to contain a line that begins with the text 'From', then the alteration of the data may well result in the file not decoding correctly.

Using consistent overhead byte stuffing, it would be easy to find the beginning of each mail message simply by looking for the reserved framing character, and no mail messages would be damaged in the process.

7.6.3 Backup Media

Many popular kinds of 'append-only' media exist, such as recordable ('write-once') CDs [ISO10149] [ECMA-167] [ECMA-168], and streaming tape formats like 4mm [X3.203] ('DAT' or 'DDS') and 8mm [ISO11319] helical-scan tapes. Adding new data to these media in anything other than a simple linear sequential fashion is cumbersome at best, and in many cases impossible. Many common backup formats have magic data in magic places, such as a catalog of files at the start of the tape. This makes it difficult to add new data to an existing backup tape because there is no convenient way to go back to the start of the tape and rewrite the catalog. It also means that an otherwise small amount of damage at the start of the tape which renders that catalog unreadable may make the entire archive effectively useless [Zwi91a] [Zwi91b] [Zwi97].

An alternative approach would be to write each file, along with its meta-data, as a self-describing record, framed using COBS. Additional meta-data, as required by the backup software, would also be written as self-describing records and framed using COBS. Framing the data in independent records this way would allow the tape reader to decode each complete undamaged record in isolation, regardless of whether the preceeding or following records were readable, and a small isolated error would not render the entire remainder of the tape meaningless.

7.7 Allowing New Observers to Join

Sometimes an observer on a network may want to observe a transmission that is already part-way through. One example of this is digital satellite broadcasting, which broadcasts a continuous MPEG video stream. For observers to join in the middle of a transmission, they must be able to detect where the next frame or packet of data begins.

Using consistent overhead byte stuffing, an observer could join a transmission at any point and still easily be able to detect the start of the next frame.

Chapter Eight

Conclusions

COBS is a useful addition to our arsenal of techniques for data communications. It is computationally cheap, easy to implement in software, 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 one kilobyte 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 to the device 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.

COBS has the potential to set a new standard, not only for packet radio applications, but for all future applications that use byte stuffing.

The benefit of conventional two-for-one substitution 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 header compression techniques exist to reduce the overhead of the packet headers (especially over slow links) [Deg96], those header compression techniques 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.

Chapter Nine

Bibliography

[Abr72] Norman Abramson. *Excess capacity of a slotted aloha channel.* ARPANET Satellite System Note 26 12735, Stanford Research Institute, November 1972.

[Abr85] Norman Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory,* vol. IT-31, pp. 119-123, March 1985.

[Ahl96a] B. Ahlgren, P. Gunningberg & K. Moldeklev. Increasing Communication Performance with a Minimal-Copy Data Path Supporting ILP and ALF. *Journal of High Speed Networks,* 5(2):203-214, 1996.

[Ahl96b] B. Ahlgren, M.Björkman & P.Gunningberg. Towards Predictable ILP Performance-Controlling Communication Buffer Cache Effects. *The Australian Computer Journal,* 28(2):66-71, May 1996.

[Ahl96c] B. Ahlgren, M. Björkman & P. Gunningberg. Integrated Layer Processing Can Be Hazardous to Your Performance. *IFIP Protocols for High Speed Networks,* Sophia-Antipolis, France, October 1996.

[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.

[Bar53] R. H. Barker. Group Synchronizing of Binary Digital Systems, *Communication Theory (Proceedings of the Symposium on the Applications of Communication Theory, September 1952)*. W. Jackson, Ed. New York: Academic Press, 1953.

[Bea90] K. G. Beauchamp. *Computer Communications, Second edition*. London, UK: Chapman and Hall, 1990.

[Braun94] H. W. Braun and K. Claffy. Web Traffic Characterization: an assessment of the impact of caching documents from NCSA's web server. *Proceedings of Second International World Wide Web (WWW) Conference '94.* Chicago, IL, October 1994.

[Braun96] T. Braun & C. Diot. Automated Code Generation for Integrated Layer Processing. *IFIP workshop on Protocol for High Speed Networks.* Sophia Antipolis (FRANCE), October 1996.

[Brit56-97] Baudot, Jean-Maurice-Émile. *Britannica Online.* Available http://www.eb.com:180/cgi-bin/g?DocF=micro/56/65.html [25 March 1997].

[Brit405-97] Morse Code. *Britannica Online.* Available http://www.eb.com:180/cgi-bin/g?DocF=micro/405/52.html [25 March 1997].

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

[Carl97] Personal communication, James Carlson, Principal Software Engineer, IronBridge Networks, September 1997.

[Che96] 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 and Mary Baker. Consistent Overhead Byte Stuffing. *Proceedings of ACM SIGCOMM 1997*, September 1997.

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

[Cro08] Arthur Crotch. *The Hughes and Baudot Telegraphs*. London: Rentell, 1908.

[Cru78] Frank da Cruz. *Kermit, A File Transfer Protocol*. Bedford, MA: Digital Press, c1987.

[Deg96] Mikael Degermark, Mathias Engan, Björn Nordgren, Stephen Pink. Low-Loss TCP/IP Header Compression for Wireless Networks. *Proceedings of MobiCom '96,* Rye, New York, November 10-12, 1996.

[Dijk74] Edsger Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM,* vol. 17, 1974, pp. 643-644.

[Dijk86] Edsger Dijkstra. A Belated Proof of Self-Stabilization. *Distributed Computing,* vol. 1, pp. 5-6, 1986.

[East65] W. L. Eastman. On the Construction of Comma-Free Codes. *IEEE Transactions on Information Theory,* vol. IT-11, pp. 263-266, April 1965

[ECMA-40] European Computer Manufacturers Association Standard ECMA-40: HDLC Frame Structure, December 1973, September 1976 & December 1979.

[ECMA-167] European Computer Manufacturers Association Standard ECMA-167: Volume and File Structure for Write-Once and Rewritable Media Using Non-Sequential Recording for Information Interchange, 3rd edition, June 1997.

[ECMA-168] European Computer Manufacturers Association Standard ECMA-168: Volume and File Structure for Read-Only and Write-Once Compact Disk Media for Information Interchange, 2nd edition, December 1994.

[For88] Chuck Forsberg. *XMODEM/YMODEM Protocol Reference*, October 1988. Available from Omen Technology Inc., Post Office Box 4681, Portland, Oregon 97208. Tel. 503-614-0430. Fax 503-629-0665. Also available http://pauillac.inria.fr/~doligez/zmodem/ymodem.txt [26 March 1997].

[For96] Chuck Forsberg. *Getting the Most out of ZMODEM*, July 1996. Available from Omen Technology Inc., Post Office Box 4681, Portland, Oregon 97208. Tel. 503-614-0430. Fax 503-629-0665. Also available http://agora.rdrop.com/users/caf/zmdmget.html [26 March 1997].

[G721] CCITT Recommendation G.700-G.795 (1988), General Aspects of Digital Transmission Systems, Terminal Equipment. ('Blue Book')

[Gill96] Personal communication, John Gill, Associate Professor, Electrical Engineering, Stanford University, November 1996. Not published in any public forum.

[Gol58a] Golomb, Welch & Delbrück. Construction and Properties of Comma-Free Codes, *Biologiske Meddelelser udgivet af Det Kongelige Danske Videnskabernes Selskab,* vol. 23, no. 9, 1958.

[Gol58b] S. W. Golomb, B. Gordon, and L. R. Welch. Comma-Free Codes. *The Canadian Journal of Mathematics*, Vol. 10, 1958.

[Gol63] S. W. Golomb, Baumert et al. Recent Results on Comma-Free Codes. *The Canadian Journal of Mathematics*, Vol. 15, 1963.

[Gwert96] James Gwertzman and Margo Seltzer. World-Wide Web Cache Consistency. *Proceedings of USENIX, 1996.* January 1996.

[Held91] Gilbert Held. *Understanding Data Communications: From Fundamentals to Networking*. Chichester, England; New York: Wiley, c1991.

[Helg91] Hermann J. Helgert. *Integrated Services Digital Networks: Architectures, Protocols, Standards*. Reading, Mass.: Addison-Wesley, 1991.

[Hol95] Gerard J. Holzmann. *The Early History of Data Networks*. Los Alamitos, Calif.: IEEE Computer Society Press, 1995.

[Hop86] Andrew Hopper, Steven Temple, and Robin Williamson. *Local Area Network Design.* Workingham, England; Reading, Mass.: Addison-Wesley, c1986.

[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. *Communication Theory (Proceedings of the Symposium on the Applications of Communication Theory, September 1952).* W. Jackson, Ed. New York: Academic Press, 1953.

[IEEE802.3] Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. Institute of Electrical and Electronic Engineers, IEEE Standard 802.3-1990, 1990.

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

[ISO 8824] Information Processing Systems — Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1), Geneva, March 1988.

[ISO 8825] Information Processing Systems — Open Systems Interconnection, Specification of Basic Encoding Rules for Abstract Notation One (ASN.1), Geneva, March 1988.

[ISO10149] ISO Committee Draft 10149. Information technology — Data interchange on read-only 120 mm optical data disks (CD-ROM), ISO/IEC 10149, 1989. (‘Yellow Book’)

[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.

[ISO11319] ISO Committee Draft 11319. Information technology — 8 mm wide magnetic tape cartridge for information interchange — Helical scan recording, ISO/IEC 11319, 1993.

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

[Kahl86] Peter Kahl. *ISDN: The Future Telecommunication Network of the Deutsche Bundespost*. Heidelberg: R. v. Decker, 1986.

[Karn90] P. Karn. MACA — A New Channel Access Method for Packet Radio. *ARRL/CRRL Amateur Radio 9th Computer Networking Conference.* London, Ontario, Canada. September 1990.

[Kend62] W. B. Kendall & I. S. Reed. Path Invariant Comma-Free Codes. *IRE Transactions on Information Theory,* vol. IT-8, pp350-355, October 1962.

[Kent87] C. Kent and J. Mogul. Fragmentation Considered Harmful. *Proc. of SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology.* August, 1987.

[Kes96] Gary C. Kessler, Peter V. Southwick. *ISDN: Concepts, Facilities, And Services. 3rd ed*. New York: McGraw-Hill, c1997 [i.e. 1996].

[King83] Richard Allen King. *The IBM PC-DOS Handbook, 1st ed.* Berkeley, CA: Sybex, 1983.

[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.

[Mauf97] Personal communication, Tom Maufer, T-1 engineer, 3 Com, March 1997.

[Mav03] William Maver, Jr. *American Telegraphy and Encyclopedia of the Telegraph: Systems, Apparatus, Operation. 5th ed.* New York: Maver publishing company [1903].

[May96] Personal communication, Maynard Handley, chief MPEG engineer, Apple Computer, November 1996.

[Mil87] Michael J. Miller, Syed V. Ahamed. *Digital Transmission Systems and Networks*. Rockville, MD: Computer Science Press, c1987.

[Nel96] Mark Nelson, Jean-Loup Gailly. *The Data Compression Book, 2nd ed.* New York: M&T Books, c1996.

[Pad83] M. A. Padlipsky. A Perspective on the ARPANET Reference Model. *Proceedings of IEEE INFOCOM '83*, p. 242-53. Silver Springs, MD: IEEE Computer Society Press.

[Pap87] Stavros Papastavridis. On the Mean Number of Stuffing Bits in a Prefix-Synchronized Code, *IEEE Transactions on Communications,* vol. COM-35, pp. 453-455, April 1987.

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

[Pra97] Personal communication, Vaughan Pratt, August 1997.

[RoFlo94] A. Romanow and S. Floyd. Dynamics of TCP Traffic over ATM Networks. *Proceedings of ACM SIGCOMM 1994*, August 1994.

[RFC791] J. Postel. Internet Protocol. STD 5, RFC 791, September 1981.

[RFC916] G. Finn. Reliable Asynchronous Transfer Protocol (RATP). RFC 916, October 1984.

[RFC935] J. Robinson. Reliable Link Layer Protocols. RFC 935, January 1985.

[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.

[RFC1122] R. Braden. Requirements for Internet Hosts — Communication Layers. RFC 1122, October 1989.

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

[RFC1191] J. Mogul and S. Deering. Path MTU Discovery. RFC 1191, November 1990.

[RFC1483] Juha Heinanen. Multiprotocol Encapsulation over ATM Adaptation Layer 5. RFC 1483, July 1993.

[RFC1661] William Simpson. The Point-to-Point Protocol (PPP). RFC 1661, July 1994.

[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.

[RS-232-C] Electronic Industries Association Standard RS-232-C: Interface between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange, October 1969.

[Sch69a] Robert A. Scholtz. On the Generation and Generalization of Comma-Free Codes, *1969 International Symposium on Information Theory*, Ellenville, New York, January 28-31, 1969.

[Sch69b] Robert A. Scholtz. Maximal and Variable Word-Length Comma-Free Codes, *IEEE Trans. on Information Theory*, Vol. IT-15, March 1969.

[Sch80] Robert A. Scholtz. Frame Synchronization Techniques, *IEEE Transactions on Communications,* vol. COM-28, pp. 1204-1213, August 1980.

[Sext92] Mike Sexton and Andy Reid. *Transmission Networking: SONET and the Synchronous Digital Hierarchy.* Boston: Artech House, c1992.

[Shah93] Amit Shah, G. Ramakrishnan. *FDDI : A High Speed Network.* Englewood Cliffs, N.J.: PTR Prentice Hall, 1993.

[Sta94] William Stallings, *Data and Computer Communications, Fourth Edition.* New York: Macmillan Pub. Co.; Toronto: Collier Macmillan Canada; New York: Maxwell Macmillan International, c1994.

[Sta95] William Stallings, *ISDN and Broadband ISDN with Frame Relay and ATM. 3rd ed.* Englewood Cliffs, N.J.: Prentice Hall, 1995.

[T1.601] ANSI Standard Number T1.601-1988. *Integrated Services Digital Network (ISDN) Basic Access Interface For Use On Metallic Loops For Application On The Network Side Of The NT (Layer 1 Specification)*. New York, N.Y.: ANSI, 1988.

[Tan88] Andrew S. Tanenbaum, *Computer Networks, 2nd ed*. Englewood Cliffs, N.J.: Prentice-Hall, c1988.

[Tay97] Tom Taylor, discussion on the IETF PPP Working Group Mailing List, 1st December 1997.

[Thom97] Kevin Thompson, Gregory J. Miller, and Rick Wilde. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network.* November/December 1997.

[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.

[Vail1845] Alfred Vail. *The American Electro Magnetic Telegraph: with the Reports of Congress, and a Description of all Telegraphs Known, Employing Electricity or Galvanism.* Philadelphia: Lea & Blanchard, 1845.

[Vei83] Richard H. Veith, *Television's Teletext.* New York, N.Y.: North-Holland, c1983.

[Ver86] Verilink Corporation, Sunnyvale, California, Interface Method and Apparatus. U.S. Patent number 4,587,514. Expired May 11, 1994 due to failure to pay maintenance fees.

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

[Wolv84] Van Wolverton. *Running MS DOS: the Microsoft guide to getting the most out of the standard operating system for the IBM PC and 50 other personal computers.* Bellevue, Wash.: Microsoft Press; [New York, N.Y.]: Distributed in the U.S. and Canada by Simon and Schuster, c1984.

[X3.131] ANSI Standard Number X3.131-1986. *Small Computer System Interface (SCSI).* New York, N.Y.: ANSI, 1986.

[X3.203] ANSI Standard Number X3.203:1992. *Helical-Scan Digital Computer Tape Cartridge, 3.81 mm (0.150 in) — Digital Data Storage (DDS) Recorded Format for Information Interchange*. New York, N.Y.: ANSI, 1992.

[Zwi91a] Elizabeth D. Zwicky. Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not. *Proceedings of the 5th Conference on Large Installation Systems Administration (LISA91)* 181-190. San Diego, CA, USA, September 1991. USENIX Association, 1991.

[Zwi91b] Elizabeth D. Zwicky. Analysing Backup Systems. *Proceedings of the 9th Annual Sun User Group Conference and Exhibit (SUG91)*. San Jose, CA, USA, December 1991. 287-300.

[Zwi97] Personal communication, Elizabeth D. Zwicky, Technical lead of the European Desktop Project, Silicon Graphics, December 1997.

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