PPP Working Group   J. Carlson
Internet Draft   IronBridge Networks
expires in six months   S. Cheshire
                                                       M. Baker
                                             Stanford University
                                                November 1997

PPP Consistent Overhead Byte Stuffing (COBS)
<draft-ietf-pppext-cobs-00.txt>

Status of this Memo

   This document is the product of the Point-to-Point Protocol
   Extensions Working Group of the Internet Engineering Task Force
   (IETF). Comments should be submitted to the ietf-ppp@merit.edu
   mailing list.

   Distribution of this memo is unlimited.

   This document is an Internet-Draft. Internet-Drafts are working
   documents of the Internet Engineering Task Force (IETF), its areas,
   and its working groups. Note that other groups may also distribute
   working documents as Internet-Drafts.

   Internet-Drafts are draft documents valid for a maximum of six
   months. Internet-Drafts may be updated, replaced, or obsoleted by
   other documents at any time. It is not appropriate to use Internet-
   Drafts as reference material or to cite them other than as a
   ``working draft'' or ``work in progress.''

   To learn the current status of any Internet-Draft, please check the
   1id-abstracts.txt listing contained in the Internet-Drafts Shadow
   Directories on ds.internic.net, nic.nordu.net, ftp.nisc.sri.com, or
   munnari.oz.au.

Abstract

   The Point-to-Point Protocol (PPP) [1] provides a standard method for
   transporting multi-protocol datagrams over point-to-point links.

   PPP also defines an extensible Link Control Protocol, which allows
   the negotiation of optional frame encoding methods. This document
   defines the Consistent Overhead Byte Stuffing (COBS) negotiation and
   encapsulation procedure.

Carlson   expires May 1998   [Page 1]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

Table of Contents

1. **Introduction**

   Standard PPP encapsulation on an asynchronous link uses an encapsula-
   tion procedure called AHDLC, and on a Synchronous Optical Network
   (SONET) or Synchronous Digital Heirarchy (SDH) link an encapsulation
   procedure called Octet Synchronous [2]. These procedures are easy to
   implement, require only a single character buffer and have good
   error-recovery characteristics, but they have a worst case expansion
   ratio of 100% where the user data consists of only hex 7D or 7E.

   This draft describes a new encapsulation method for PPP due to origi-
   nal work by Stuart Cheshire and Mary Baker at the Stanford University
   Computer Science Department [3]. This new method is slightly more
   complex than either of the two standard encodings and requires a 207
   character buffer, but it has the same error-recovery characteristics
   and has a worst-case expansion of less than 0.5%.

   This low bound on worst-case expansion has a number of benefits. For
   applications requiring Quality of Service (QoS) guarantees, it may be
   necessary to over-provision a line using PPP by a factor of two in
   order to deliver the requested bandwidth or to suffer possible queu-
   ing delays when data do expand. For applications where the underly-
   ing transport has message size limits, such as some radio protocols,
   conventional PPP byte stuffing requires that the PPP Maximum Receive
   Unit (MRU) and the associated network Maximum Transmission Units

Carlson   expires May 1998   [Page 2]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

   (MTUs) be reduced to half of the underlying hardware MTU. Even where
   these considerations are not important, COBS options can provide
   lower overhead than standard encoding methods, resulting in better
   performance.

   It should be noted that this concern over pathological data patterns
   which double in size is not entirely academic. Certain protocols,
   such as raw Pulse-Code Modulated (PCM) voice over User Datagram Pro-
   tocol (UDP), can be prone to sending an excessive density of 7E char-
   acters which will cause the standard encapsulations to double the
   amount of actual data sent. Ironically, these protocols are pre-
   cisely the ones that are likely to want guaranteed bandwidth ser-
   vices. COBS avoids those expansion cases entirely because its
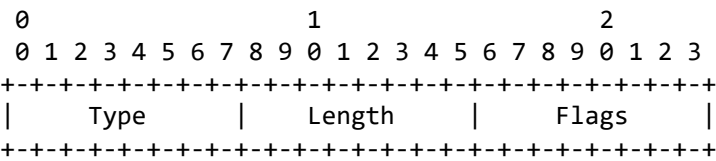   worst-case expansion in encoded data is less than 0.5%.

1.1. Conventions

   The following language conventions are used in the items of specifi-
   cation in this document:

     o MUST, SHALL, or MANDATORY -- This item is an absolute require-
        ment of the specification.

     o SHOULD or RECOMMEND -- This item should generally be followed
        for all but exceptional circumstances.

     o MAY or OPTIONAL -- This item is truly optional and may be fol-
        lowed or ignored according to the needs of the implementor.

2. COBS Configuration Option Format

   A summary of the COBS Configuration Option format for the Link Con-
   trol Protocol (LCP) is shown below. The fields are transmitted from
   left to right.

```
    0                   1                   2
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     Type      |    Length     |     Flags     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

   Type

      To Be Determined

Carlson   expires May 1998   [Page 3]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

     Length

        3

     Flags

        The flags are a single octet representing options which are passed
        from the receiver to the transmitter. The most significant six
        bits of this octet are reserved, and MUST be set to zero on
        transmit and ignored on reception.

            0   1   2   3   4   5   6   7
         +-----+-----+-----+-----+-----+-----+-----+-----+
         | Res | Res | Res | Res | Res | Res | PRE | ZXE |
         +-----+-----+-----+-----+-----+-----+-----+-----+

        These flags are:

           ZXE   If set to 1, then the receiver supports both zero pair
                  elimination (ZPE) and zero run elimination (ZRE).

           PRE   If set to 1, then the receiver supports packet preemp-
                  tion, which allows the sender to interrupt a COBS packet
                  in mid-stream to send a higher priority packet, and to
                  then return to the lower priority data.

     This option is a boolean flag appearing at most once in a single
     Configure-Request message. If it is present in a Configure-Request
     message, then the sender wishes to receive data encoded using COBS
     once LCP reaches Open state. If it is absent from the request, then
     the sender does not wish to receive COBS data. If the option is
     included in a Configure-Reject, then the sender is unable to transmit
     using COBS and will continue to use the standard method for this link
     (either Octet Synchronous or AHDLC) when LCP reaches the Open state.

     The value of the Flags field is not negotiated because all COBS
     transmitters SHALL be capable of handling all possible flag combina-
     tions. Just as with MRU negotiation, where a receiver indicating
     that it is prepared to receive packets up to 2048 octets in length
     does not obligate its peer to actually send any packet with as many
     as 2048 octets, a receiver indicating that it is prepared to receive
     either of the optional COBS encodings does not obligate its peer to
     actually send any packet using those encodings. Consequently, the
     COBS option MUST NOT be included in a Configure-Nak message in reply
     to a Configure-Request containing this option, even if the peer does
     not implement some or all of the options that the receiver has indi-
     cated its willingness to receive.

Carlson   expires May 1998   [Page 4]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

An implementation that requires the use of COBS for normal operation
MAY, however, choose to send an "unsolicited" Configure-Nak with the
COBS option if the peer fails to include the COBS option in its
Configure-Request.

3. Encapsulation Method

Like the standard encapsulations, COBS uses an octet of hex 7E to
mark the bounds between frames. The value 7E does not appear in the
frame data itself. Using the value 7E helps ensure compatibility
with any existing software and hardware that may assume that the PPP
frame boundary marker is always 7E. It also aids recovery in the
case where errors occur and the transmitter and receiver are not in
agreement about whether COBS encapsulation is in effect. Because the
framing marker is the same regardless of whether COBS encapsulation
is in effect, frame boundaries are still detected correctly, and this
allows the error recovery described in section 3.5 to work very
easily.

COBS encapsulation is a simple reversible transformation that elim-
inates all instances of hex 7E from the frame to be transmitted.
This COBS encoding procedure is logically a two-step process,
although in real implementations both steps are performed in a single
loop for the sake of efficiency. The first step ("zero elimination")
eliminates all occurrences of zeroes from the data, while guarantee-
ing to add at most no more than 0.5% to the data size. This results
in a data packet containing only byte values hex 01 to hex FF, and no
zeroes. The second step ("7E substitution") replaces all occurrences
of hex 7E with hex 00, thereby producing a packet that does contain
zeroes but contains no instances of hex 7E.

The zero elimination step encodes any data packet using a series of
COBS code blocks. Each COBS code block begins with a single code
byte, followed by zero or more data bytes. The code byte determines
how many data bytes follow. The codes and their meanings are deter-
mined such that all possible data packets can be encoded as a valid
series of code blocks, and furthermore, even in the worst possible
case, there exists no valid encoding that adds more than 0.5% over-
head to the packet size. There is no pre-set limit to the length of
packet that may be encoded. The value zero is never used as a code
byte, nor does it ever appear as a data byte, which is why the output
of COBS zero elimination never contains any instances of the value
zero.

The 7E substitution step allows a linear code range to be used for
octet counts without concern for the potential end-of-frame marker in
the middle of the code space.

Carlson   expires May 1998   [Page 5]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

   The PPP/COBS codes and their meanings are listed below:

   Code (n) Followed by: Meaning
   -------- ------------   -------
    00   Unused (framing character placeholder)
    01-CF   n-1 data bytes   The data bytes, plus implicit trailing zero
    D0   n-1 data bytes   The data bytes, no implicit trailing zero
    D1   Unused (resume preempted packet)
    D2   Unused (reserved for future use)
    D3-DF   nothing   a run of (n-D0) zeroes
    E0-FE   n-E0 data bytes The data bytes, plus two trailing zeroes
    FF   Unused (PPP error)

   Code byte hex 00 is never used, in order to provide the required
   "zero elimination" property.

   Code byte hex D1 is never used (although value D1 may appear as a
   data byte). If a COBS receiver observes that the first byte after a
   framing marker is value D1, then it means that this new "packet" of
   PPP data resumes transmission of a previously preempted packet (see
   section 3.4).

   Code byte hex D2 is never used (although value D2 may appear as a
   data byte). Code byte hex D2 is reserved for future use.

   Code byte hex FF is never used (although value FF may appear as a
   data byte). If a COBS receiver observes that the first byte after a
   framing marker is value FF, then this indicates that an error has
   occurred (see section 3.5).

   When negotiated, COBS goes into effect when LCP reaches the Open
   state. When COBS is in effect, subsequent frames, including LCP mes-
   sages such as Protocol-Reject, must be sent using COBS. If LCP
   leaves the Open state, then COBS must be disabled. If LCP is renego-
   tiated or if the peer is restarted, then COBS may be disabled
   silently; this is detected by the procedure in section 3.5.

   An implementation SHOULD disable COBS transmission before sending an
   LCP Terminate-Request message.

3.1. Frame Transmission

   As with AHDLC and Octet Synchronous encoding, a 7E is used to mark
   the frame boundary. This value is guaranteed never to occur within
   COBS encoded data.

   The COBS zero elimination procedure effectively searches the packet

Carlson   expires May 1998   [Page 6]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

for the first occurrence of value zero. To simplify the encoding
procedure, all packets are treated as though they end with a trailing
zero at the very end, after the standard CRC. This "phantom" octet
of hex 00 is automatically discarded after decoding at the receiving
end to correctly reconstruct the original packet data.

The number of octets up to and including the first zero determines
the code to be output. If this number is 207 or fewer, then this
number is output as the code byte, followed by the actual non-zero
bytes themselves. The zero is skipped and not output; the receiver
will automatically add the zero back in as part of the decoding pro-
cess. If there are 207 or more non-zero bytes, then code hex D0 is
output, followed by the first 207 non-zero bytes. This process is
repeated until all of the bytes of the packet (including the phantom
trailing zero at the end) have been encoded.

As an optional optimization, if the receiver has indicated its desire
to receive zero-pair and zero-run codes in its COBS Configuration
Option, then the transmitter MAY elect to encode zero pairs and zero
runs more efficiently. If a pair of 00 octets are found in the input
data after 0 to 30 non-zero octets, then the count of non-zero octets
plus E0 is output, followed by the non-zero octets, and both 00
octets in the input data are skipped. If a run of three to fifteen
00 octets are found in the input data, then the count of these 00
octets plus D0 is output and the 00 octets in the input data are
skipped. See section 3.3 below for more details.

If the receiver has indicated that it supports packet preemption and
the sender is also configured to support it, then it is possible to
preempt the state of the current COBS packet within two bytes and
send another. If preemption is required for a high-priority packet,
then the output state is checked. If the output is idle (no COBS
output is in progress) then the packet is sent normally. If the out-
put is busy (at least one data octet has been output for a packet in
progress), then an error is forced to interrupt the current packet
and the current packet being transmitted is saved but the COBS encod-
ing state is discarded, and the high-priority packet is then sent
using COBS encoding. When the high-priority packet is complete and
the terminating 7E has been sent, the sender MAY resume the saved
packet by issuing a D1 code after the terminating 7E, and then res-
tarting COBS encoding. If a subsequent high-priority packet requires
transmission instead, then it MAY be sent immediately. See section
3.4 below for more details.

Note that both of these options are proper supersets of the basic
encoding, so a transmitter that does not support an option which the
receiver does support will always send data that the receiver can
correctly decode. Likewise, if the receiver does not support a given

Carlson   expires May 1998   [Page 7]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

     option, the transmitter MUST disable its use of this encoding. This
     arrangement allows the LCP negotiation for COBS to be quite simple.

     The COBS count code FF is never used. Since a COBS count always fol-
     lows a 7E, this means that a COBS encoder will never generate 7E FF.
     This sequence is instead used to disable COBS in the event that LCP
     is renegotiated; see section 3.5.

3.2. Frame Reception

     The frame boundaries are located by a 7E as with AHDLC and Octet Syn-
     chronous encodings. A frame boundary in the input stream terminates
     the decoding process. If the boundary occurs at the end of a COBS
     code block then the packet is deemed complete and the final trailing
     ("phantom") zero is removed before the packet is passed to the higher
     layer for processing. If the boundary occurs within a COBS code
     block then this is an error or, if the receiver supports it, a poten-
     tial packet preemption. See section 3.4 below for more details on
     packet preemption.

     If the first octet of the frame is hex FF, then recovery is attempted
     as in section 3.5. Otherwise, the stream of octets within a within a
     frame is transformed by converting all instances of 00 to 7E. If the
     first octet of the frame is hex D1, then this "packet" resumes
     transmission of a previously preempted packet. See section 3.4 below
     for more details on packet preemption. Otherwise, the COBS decoder
     decodes the encoded byte stream as described below:

     The COBS decoder reads the first byte (the code byte).

     If the octet is D0, then 207 octets of data are copied from the input
     stream to the frame, and no 00 octets are appended.

     Otherwise, if the most significant four bits are 1101, and ZPE/ZRE is
     implemented in the receiver, then the least significant four bits are
     used as a count of 00 octets to append to the frame data. If ZPE/ZRE
     is not implemented in the receiver, then this is an error.

     Otherwise, if the most significant three bits are 111, and ZPE/ZRE is
     implemented in the receiver, then this encodes a run followed by a
     pair of zeroes. The least significant five bits are used as a count
     of octets to read from the input data stream. These octets are
     copied to the frame being decoded, and then two 00 octets are
     appended to the frame data. If ZPE/ZRE is not implemented in the
     receiver, then this is an error.

     Otherwise, the value of the octet is a counter. Count-1 octets of

Carlson   expires May 1998   [Page 8]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

    the input stream are copied to the frame and a single 00 octet is
    appended.

    The decoding process then continues by reading the next code byte
    from the input and repeating the decoding process described above
    until all bytes of the input have been consumed.

3.3. Zero-pair and zero-run encoding

    The goal of COBS encoding is to provide a standard that is suitable
    for use on both the highest speed links and the lowest speed links.

    One dilemma that faces designers of higher level protocols today is
    that for efficiency at very high speeds it is beneficial to align
    fields to 64-bit or even 128-bit boundaries. The IPv6 header format
    is an example of this trend. Unfortunately this can often mean
    inserting padding zeroes between fields to ensure proper alignment,
    thereby wasting precious bandwidth when these packets are sent over
    low speed links. Another common practice is for protocol designers
    to define fields that are reserved for future expansion, accompanied
    by the familiar phrase "MUST be set to zero on transmission and
    ignored on reception." These extra unnecessary zeroes in packets also
    waste bandwidth.

    The purpose of COBS's zero-pair and zero-run encoding is to remove
    this protocol designers' dilemma. COBS's zero-pair and zero-run
    encoding can "compress out" block of zeroes from packet data, thereby
    making the cost of these extra zeroes negligible. This allows proto-
    col designers to design one single format for their protocol, without
    being forced to choose between either favoring high speed or favoring
    low speed links.

    Particularly for string fields, it can be convenient to have a fixed
    size field that is considerably longer than the typical string length
    it holds, in order to keep a simple fixed-format packet structure
    while at the same time not unduly limiting the length of string that
    may be used in that field. As long as the unused bytes are zero-
    filled, zero-pair and zero-run encoding will "compress out" those
    unused bytes, making their cost negligible.

    Zero-pair and zero-run encoding are not intended to compete with more
    sophisticated (and more computationally costly) compression algo-
    rithms such as Lempel-Ziv [4] or Huffman [5] encoding. On the very
    slowest links, these compression algorithms may still be appropriate.
    The highest compression ratios are achieved by algorithms such as Van
    Jacobson's TCP header compression which take advantage of interdepen-
    dencies between packets. However, this means that a single packet

Carlson   expires May 1998   [Page 9]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

        loss on the link can cause multiple packets to be unrecoverable,
        which may not be appropriate for technologies such as wireless links
        where packet loss rates are already relatively high.

        One aspect of zero-run compression is that on reception this
        compressed data has to be expanded back to its original size. On the
        very highest speed links, where the link data rate is comparable to
        the memory system bandwidth, this potential 15:1 expansion at the
        receiver may be unacceptable. For this reason, the receiver may
        request in its COBS Configuration Option that the sender *not* gen-
        erate zero-pair (hex E0-FE) or zero-run (hex D3-DF) COBS codes. If
        the receiver does not indicate its desire to receive zero-pair and
        zero-run codes in its COBS Configuration Option, the transmitter MUST
        not generate these codes.

        The reverse situation -- of a COBS transmitter that does not imple-
        ment zero-pair and zero-run encoding -- needs no negotiation. Even
        if the receiver indicates its willingness to receive zero-pair and
        zero-run codes of a transmitter that does not implement them, the
        transmitter's output, while not containing those codes, will still be
        a perfectly legal encoding of the packets. It may be a sub-optimal
        encoding, but it is still a legal encoding that the receiver will
        decode correctly.

 3.4. Packet Preemption

        On low speed links there can be conflicting goals. To provide effi-
        ciency, it is desirable to allow packets to be as large as possible.
         However, if a small high-priority packet arrives just as a large
        low-priority packet begins transmission, the high-priority packet is
        delayed until transmission of the large low-priority packet is com-
        plete, which favors making the maximum packet size relatively small.
        Packet preemption solves this dilemma by allowing the link to suspend
        transmission of a low-priority packet immediately whenever a high-
        priority packet needs to be sent, and then resume transmission of the
        low-priority packet afterwards.

        COBS packet preemption allows us to do this with at most two bytes of
        additional overhead compared to sending the packets in the normal
        sequential manner.

        If the receiver has indicated in its COBS Configuration Option that
        it supports packet preemption, then the transmitter can preempt a
        low-priority packet at any time simply by forcing an error and then
        beginning transmission of the high-priority packet. After the high-
        priority packet(s) has (have) been sent, the sender resumes transmis-
        sion of the preempted packet by transmitting a code hex D1 (resume

Carlson   expires May 1998   [Page 10]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

preempted packet) and then resuming COBS encoded transmission from
the first untransmitted byte of the preempted packet.

During packet transmission, the transmitter is either in a state in
which it is transmitting data within a COBS block or it has just sent
the last octet of a COBS block and is preparing to send the code byte
to start a new COBS block. To force an error for packet preemption,
a 7E code must be sent within the data portion of a COBS block.
Therefore, if the transmitter is preparing to send a new counter when
preemption is necessary, it should either calculate and send that
counter first, or it should send a dummy counter of 02. In any case,
the 7E that follows will signal the preemption. The receiver will
detect an incomplete COBS code block as an error.

A receiver that supports packet preemption must maintain two packet
receive buffers. The two buffers are equal in status, but at any
point in time one buffer is the "active buffer" and the other is the
"inactive buffer". If an error occurs in the course of COBS decod-
ing, then the receiver treats it as a possible indication of packet
preemption. The receiver remembers the number of decoded bytes that
have been written into the active buffer, and the other buffer now
becomes the active buffer. The receiver then proceeds to receive
packets as normal into this buffer. When the receiver observes a
packet that begins with the byte value hex D1, it recognizes this as
an indication to resume a previously preempted packet. The active
and inactive buffers are again swapped, but now instead of beginning
writing to the start of the buffer, the receiver proceeds to append
bytes after any data that is already there. In the unlikely event
that the inactive buffer contains no bytes of data, this is not con-
sidered an error. A previous packet may have been preempted before
even a single byte of data was decoded, in which case having zero
bytes of data already in the buffer when the packet resumes is in
fact correct.

The following example shows a large packet preempted by two small
ones. The large packet contains the hex values "01 02 03 04 05 06
07" and the two small packets contain the values "11 12 13" and "21
22 23" respectively. If the large packet is preempted after three
bytes, the correct encoding of this data is:

7E 08 01 02 03 7E 04 11 12 13 7E 04 21 22 23 7E D1 05 04 05 06 07 7E

The byte-stream begins with the framing marker 7E. The encoder then
indicates that it plans to send seven non-zero bytes (COBS code hex
08). It then sends the first three of those non-zero bytes (01 02
03) before a pair of high-priority packets arrive. The sender then
interrupts the transmission of the low-priority packet with another
framing marker 7E, and sends the two high-priority packets. The

Carlson   expires May 1998   [Page 11]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

receiver holds the three bytes of the partially received packet in
the inactive buffer while it is receiving the high-priority packets.
When the high-priority packets are complete, the sender sends a code
hex D1 to indicate that it is resuming transmission of the preempted
packet. The encoder indicates that it plans to send four non-zero
bytes (COBS code hex 05) and sends the remaining four bytes of the
low-priority packet.

Since packet preemption is most useful on low-speed links, high-speed
COBS implementations may elect not to implement packet preemption.
If the receiver does not indicate that is supports packet preemption
in its COBS Configuration Option, the transmitter MUST NOT preempt
packets.

The reverse situation -- of a COBS transmitter that does not imple-
ment packet preemption -- needs no negotiation. Even if the receiver
indicates its willingness to allow packet preemption, the
transmitter's output, while not containing any packet preemption,
will still be a perfectly legal steam of encoded packets.

3.5. Recovery on LCP Renegotiation

Since all implementations are required to send LCP frames with the
standard address and control fields, regardless of prior negotiation,
all LCP frames must begin with the hex sequence FF 03 C0 21 before
any byte-stuffing occurs.

The first octet of COBS data is always a counter value. Because of
the encoding methods chosen, this counter is never sent as FF. This
is to allow for recovery in case LCP is renegotiated or synchroniza-
tion is lost with the peer. Since LCP frames must begin with FF, any
frame seen when COBS is in use which begins with hex FF must
represent the start of an LCP frame without COBS enabled. The
receiver should disable COBS and revert back to Octet Synchronous or
AHDLC decoding as appropriate.

COBS speaking implementations using AHDLC MUST NOT default to escap-
ing FF when sending LCP frames unless COBS is explicitly disabled
since that would compromise this recovery mechanism.

3.6. Handling of Corrupted Data

If data are lost or corrupted during transmission, it is desirable
that the errors affect a minimum number of packets. For COBS, the
error characteristics are similar to AHDLC and Octet Synchronous
encodings.

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

     If one of the code byte octets is lost or corrupted, then the block
     will be miscounted. This is likely to ultimately result in the
     inclusion of the trailing 7E within the data portion of an erroneous
     COBS block. The receiver will detect this as either an error (if it
     does not support preemption) or as a preempted message. In the
     latter case, the falsely preempted message will be discarded when the
     next true preemption occurs. If the 7E still falls on a natural
     boundary between COBS blocks, then COBS will not detect the error,
     but the standard Frame Check Sequence (FCS) will be used to detect
     the corruption.

     If one of the data octets is corrupted, then the FCS will be used to
     detect the corruption. As above, lost data octets will result in a
     trailing 7E being included within the data portion of a message and
     result in the loss of that one packet.

     If the preemption-resume signal (D1) is lost, then the resumed data
     stream will be treated as an independent frame and will be discarded
     with an FCS error.

     If data are corrupted such as to deliver 7E FF to a COBS receiver,
     that receiver should restart LCP renegotiation due to the apparent
     loss of state and should then restart COBS. For COBS encoding to be
     viable on a given link, the probability of this type of corruption
     must be acceptably low.

     As with AHDLC and Octet-Synchronous, the loss of a 7E marker between
     frames will result in the loss of those two frames.

4. Source Code

     Two implementations are given for reference. The first implementa-
     tion is a basic version which encodes and decodes linear blocks and
     includes ZPE/ZRE. It is able to handle multiple blocks within one
     coding unit by use of multiple sequential invocations.

     The second is designed for a system using BSD-like mbufs for the PPP
     stack and using a very simple FIFO buffer interface for data transmit
     and receive. It includes standard 16-bit FCS generation and checking
     and implements both the ZPE/ZRE option and the packet preemption
     option.

4.1. Linear buffer encoding and decoding

     typedef unsigned char u_char;   /* 8 bit quantity */

Carlson   expires May 1998   [Page 13]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
    typedef enum
      {
      Unused       = 0x00,    /* Unused (framing character placeholder) */
      DiffZero     = 0x01,    /* Range 0x01 - 0xCE:                     */
      DiffZeroMax  = 0xCF,    /* n-1 explicit characters plus a zero    */
      Diff         = 0xD0,    /* 207 explicit characters, no added zero */
      Resume       = 0xD1,    /* Unused (resume preempted packet)       */
      Reserved     = 0xD2,    /* Unused (reserved for future use)       */
      RunZero      = 0xD3,    /* Range 0xD3 - 0xDF:                     */
      RunZeroMax   = 0xDF,    /* 3-15 zeroes                            */
      Diff2Zero    = 0xE0,    /* Range 0xE0 - 0xFE:                     */
      Diff2ZeroMax = 0xFE,    /* 0-30 explicit characters plus 2 zeroes */
      Error        = 0xFF     /* Unused (PPP LCP renegotiation)         */
      } StuffingCode;

    /* These macros examine just the top 3/4 bits of the code byte */
    #define isDiff2Zero(X) (((X) & 0xE0) == (Diff2Zero & 0xE0))
    #define isRunZero(X)   (((X) & 0xF0) == (RunZero   & 0xF0))

    /* Convert from single-zero code to corresponding double-zero code */
    #define ConvertZP (Diff2Zero - DiffZero)

    /* Allow generation of zero pair and zero run code blocks? */
    int ZPZR = 1;

    /* Highest single-zero code with a corresponding double-zero code */
    #define MaxConvertible (ZPZR ? Diff2ZeroMax - ConvertZP : 0)

    /* Convert to/from 0x7E-free data for sending over PPP link */
    static u_char Tx(u_char x) { return(x == 0x7E ? 0 : x); }
    static u_char Rx(u_char x) { return(x == 0 ? 0x7E : x); }

    /*
     * StuffData stuffs "length" bytes of data from the buffer "ptr",
     * writing the output to "dst", and returning as the result the
     * address of the next free byte in the output buffer.
     * The size of the output buffer must be large enough to accommodate
     * the encoded data, which in the worst case may expand by almost
     * 0.5%.  The exact amount of safety margin required can be
     * calculated using (length+1)/206, rounded *up* to the next whole
     * number of bytes.  E.g. for a 1K packet, the output buffer needs to
     * be 1K + 5 bytes to be certain of accommodating worst-case packets.
     */

    #define FinishBlock(X) \
        (*code_ptr = Tx(X), code_ptr = dst++, code = DiffZero)

    static u_char *StuffData(const u_char *ptr, unsigned int length,
```

Carlson   expires May 1998   [Page 14]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
                            u_char *dst, u_char **code_ptr_ptr)
    {
    const u_char *end = ptr + length;
    u_char *code_ptr  = *code_ptr_ptr;
    u_char code       = DiffZero;

    /* Recover state from last call, if applicable */
    if (code_ptr) code = Rx(*code_ptr);
    else code_ptr = dst++;

    while (ptr < end)
      {
      u_char c = *ptr++;  /* Read the next character */
      if (c == 0)       /* If it's a zero, do one of these operations */
        {
        if    (isRunZero(code) && code < RunZeroMax) code++;
        else if (code == Diff2Zero)                  code = RunZero;
        else if (code <= MaxConvertible)             code += ConvertZP;
        else                                         FinishBlock(code);
        }
      else             /* else, non-zero; do one of these operations */
        {
        if      (isDiff2Zero(code)) FinishBlock(code - ConvertZP);
        else if (code == RunZero)   FinishBlock(Diff2Zero);
        else if (isRunZero(code))   FinishBlock(code-1);
        *dst++ = Tx(c);
        if      (++code == Diff)    FinishBlock(code);
        }
      }
    *code_ptr_ptr = code_ptr;
    FinishBlock(code);
    return(dst-1);
    }

 /*
  * UnStuffData decodes "srclength" bytes of data from the buffer
  * "ptr", writing the output to "dst". If the decoded data does not
  * fit within "dstlength" bytes or any other error occurs, then
  * UnStuffData returns NULL.
  */
 static u_char *UnStuffData(const u_char *ptr, unsigned int srclength,
                                u_char *dst, unsigned int dstlength)
    {
    const u_char *end   = ptr + srclength;
    const u_char *limit = dst + dstlength;
    while (ptr < end)
      {
      int z, c = Rx(*ptr++);
```

Carlson   expires May 1998   [Page 15]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
      if (c == Error || c == Resume || c == Reserved) return(NULL);
      else if (c == Diff)       { z = 0;        c--;        }
      else if (isRunZero(c))    { z = c & 0xF; c = 0;       }
      else if (isDiff2Zero(c))  { z = 2;        c &= 0x1F; }
      else                      { z = 1;        c--;        }

      while (--c >= 0 && dst < limit) *dst++ = Rx(*ptr++);
      while (--z >= 0 && dst < limit) *dst++ = 0;
      }
    if (dst < limit) return(dst-1);
    else return(NULL);
    }


 /* Example showing use of chained StuffData calls */
 unsigned int StuffExample(const u_char *head, unsigned int hlength,
                           const u_char *data, unsigned int dlength,
                           u_char *dst)
   {
   u_char *ptr = dst;
   u_char *stuffstate = NULL;

   /* First stuff the packet header into the buffer */
   ptr = StuffData(head, hlength, ptr, &stuffstate);

   /* Then append the packet body to the stuffed header */
   ptr = StuffData(data, dlength, ptr, &stuffstate);

   /* Then return the total length of data generated */
   return(ptr-dst);
   }
```

4.2. PPP/COBS Encoding with mbufs

4.2.1. PPP Context Handling

```
 struct cobs_context {
    /* Transmit encoding state */
    struct txcontext {
        struct mbuf *bufs,*nextp;
        short count,zskip,lcount;
    } tx[2];
    int txpri,txendofframe,txintr;
    int txallowpri,txallowzxe;

    /* Receive decoding state */
    struct mbuf *mhead,*mtail;
```

Carlson   expires May 1998   [Page 16]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
        struct mbuf *savedpkt;
        int rxcount,rxlcount,zadd;
        u_short rxfcs;
        u_short savedfcs;
        u_char rxlchar;
    };

    /*
     * Simple context initialization.  User is responsible for setting
     * cb.txallowpri and cb.txallowzxe based on the outcome of PPP
     * negotiation.
     */
    void
    init_cobs_context(struct cobs_context *cb)
    {
        bzero(cb,sizeof(*cb));
        cb->rxfcs = 0xFFFF;
    }
```

4.2.2. PPP Frame Transmission

```
    /*
     * This is called once for each transmit-FIFO-empty interrupt.  It
     * takes a pointer to the transmit FIFO buffer and the available
     * room in that buffer, and returns the number of bytes inserted.
     * Priority is supported even if the peer doesn't support COBS
     * priority interruption.  If txallowpri is cleared, then priority
     * packets are sent out on packet boundaries rather than as
     * interrupts.
     */
    int
    ppp_cobs_xmit(struct cobs_context *cb, u_char *buffer, int buflen)
    {
        u_char *dp,chr,*obuf;
        int len,count;
        struct mbuf *mb,*mb2;
        struct txcontext *tx;

        if (buflen <= 0 || buffer == NULL || cb == NULL)
            return 0;

        /*
         * If there's a high priority packet waiting and we're in the
         * middle of sending a low priority one, then send the escape
         * flag and switch.
         */
        obuf = buffer;
```

Carlson   expires May 1998   [Page 17]

```
        if (!cb->txendofframe && cb->tx[1].nextp != NULL &&
            cb->txpri == 0) {
            if (cb->tx[0].bufs == NULL)
                cb->txpri = 1;
            else if (buflen >= 2 && cb->txallowpri) {
                cb->txpri = 1;
                if (cb->tx[0].count == 0) {
                    *buffer++ = 0x02;
                    buflen--;
                }
                *buffer++ = 0x7E;
                buflen--;
                cb->tx[0].count = cb->tx[0].zskip = cb->tx[0].lcount = 0;
                cb->txintr = 1;
                cb->tx[1].lcount = -1;
            }
        }

        tx = cb->tx + cb->txpri;
        if ((mb = tx->bufs) == NULL) {
            if ((mb = tx->nextp) != NULL) {
                tx->nextp = mb->m_act;
                mb->m_act = NULL;
            }
        }
        count = tx->count;
        while (buflen > 0) {
            if (cb->txendofframe) {
                switch (cb->txendofframe) {
                case 1:
                    /*
                     * If the zero removal falls right on the end of
                     * the packet, then we need to add a dummy code to
                     * insert a 00 byte which will be deleted by the
                     * receiver.
                     */
                    *buffer++ = 0x01;
                    buflen--;
                    cb->txendofframe = 2;
                    break;
                case 2:
                    /* Packet done; send frame mark */
                    *buffer++ = 0x7E;
                    count = 0;
                    buflen--;
                    cb->txendofframe = 0;
                    tx->lcount = -1;
                    break;
```

```
                    }
                    continue;
                }
            if (count == 0) {
                if (mb == NULL) {
                    tx->bufs = mb;
                    tx->count = count;
                    if (cb->txpri == 1 && cb->tx[0].bufs != NULL) {
                        cb->txpri = 0;
                        tx = cb->tx;
                        mb = tx->bufs;
                    } else {
                        if (cb->tx[0].nextp == NULL &&
                            cb->tx[1].nextp == NULL)
                            break;
                        cb->txpri = (cb->tx[1].nextp != NULL);
                        tx = cb->tx + cb->txpri;
                        tx->bufs = mb = tx->nextp;
                        tx->nextp = mb->m_act;
                        mb->m_act = NULL;
                    }
                    count = tx->count;
                    if (cb->txpri == 0 && cb->txintr) {
                        cb->txintr = 0;
                        *buffer++ = 0xD1; /* Resume packet */
                        buflen--;
                    }
                    continue;
                }

                /* Do look-ahead for encoding modes */
                dp = mtod(mb,u_char *);
                if (*dp == 0 && cb->txallowzxe) {
                    mb2 = mb;
                    len = mb->m_len;
                    while (mb2 != NULL && count < 15)
                        if (*dp++ == 0) {
                            count++;
                            if (--len <= 0) {
                                do {
                                    mb2 = mb2->m_next;
                                    if (mb2 == NULL)
                                        break;
                                    dp = mtod(mb2,u_char *);
                                    len = mb2->m_len;
                                } while (len <= 0);
                            }
                        } else
```

  Carlson   expires May 1998   [Page 19]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
                             break;
                    if (mb2 == NULL)
                        count++;      /* Include phantom zero byte here */
                }
            if (count >= 3) {
                tx->zskip = count;
                chr = 0xD0 + count;
                count = 0;
            } else {
                dp = mtod(mb,u_char *);
                len = mb->m_len;
                mb2 = mb;
                count = 0;
                while (mb2 != NULL && count < 207)
                    if (*dp++ != 0) {
                        count++;
                        if (--len <= 0) {
                            do {
                                mb2 = mb2->m_next;
                                if (mb2 == NULL)
                                    break;
                                dp = mtod(mb2,u_char *);
                                len = mb2->m_len;
                            } while (len <= 0);
                        }
                    } else
                        break;
                if (count == 207)
                    chr = 0xD0;
                else {
                    chr = count+1;
                    tx->zskip = 1;
                    if (count <= 30 && cb->txallowzxe &&
                        mb2 != NULL) {
                        if (len <= 1)
                            do {
                                mb2 = mb2->m_next;
                                if (mb2 == NULL)
                                    break;
                                dp = mtod(mb2,u_char *);
                                len = mb2->m_len;
                            } while (len <= 0);
                        if (mb2 == NULL || *dp == 0) {
                            chr = count + 0xE0;
                            tx->zskip = 2;
                        }
                    }
                }
            }
```

 Carlson   expires May 1998   [Page 20]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
                }
                tx->lcount = chr;
            } else {
                if (mb->m_len == 0) {
                    mb = m_free(mb);
                    continue;
                }
                chr = *mtod(mb,u_char *);
                mb->m_len--;
                mb->m_off++;
                count--;
            }
            if (chr == 0x7E)
                chr = 0;
            *buffer++ = chr;
            buflen--;
            if (count == 0) {
                count = tx->zskip;
                while (mb != NULL && count > 0) {
                    len = mb->m_len;
                    if (len > count) {
                        mb->m_len -= count;
                        mb->m_off += count;
                        count = 0;
                        break;
                    }
                    count -= len;
                    mb = m_free(mb);
                }
                tx->zskip = 0;
                while (mb != NULL && mb->m_len == 0)
                    mb = m_free(mb);
                if (mb == NULL)
                    if (count == 0 && tx->lcount != 0xD0)
                        cb->txendofframe = 1;
                    else
                        cb->txendofframe = 2;
                count = 0;
            }
        }
    }
    tx->bufs = mb;
    tx->count = count;
    return buffer-obuf;
}

/*
 * This is called once for each out-bound packet.  The arguments are
 * the COBS state structure, the pointer to the out-bound mbuf chain,
```

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
     * and the priority level of the packet (0 or 1 for low or high).
     */
    void
    ppp_cobs_enqueue(struct cobs_context *cb, struct mbuf *mb, int pri)
    {
        u_short fcs;
        u_char *dp;
        int len;
        struct mbuf *mb2;
        struct txcontext *tx;

        /*
         * It turns out to be horribly complicated to support both the
         * scan-ahead functions and the CRC calculation at the same time
         * since the last byte of the CRC (or even both bytes) may be 00.
         * Thus, it is necessary in this implementation to do the CRC
         * first and the COBS encoding second in two separate steps.  If
         * the COBS output were fed into a simple linear output buffer
         * big enough to hold the largest packet, then this could be
         * greatly simplified.
         */
        if (cb == NULL || mb == NULL)
            return;
        fcs = 0xFFFF;
        for (mb2 = mb; mb2 != NULL; mb2 = mb2->m_next) {
            dp = mtod(mb2,u_char *);
            for (len = mb2->m_len; len > 0; len--)
                fcs = (fcs >> 8) ^ fcstab[(fcs ^ *dp++) & 0xff];
        }
        mb2 = dtom(dp);
        if (mb2->m_len+mb2->m_off > MMAXOFF-2) {
            mb2->m_next = m_get(M_DONTWAIT,MT_PPPTX);
            mb2 = mb2->m_next;
            dp = mtod(mb2,u_char *);
        }
        fcs = ~fcs;
        dp[0] = fcs&0xFF;
        dp[1] = fcs>>8;
        mb2->m_len += 2;

        tx = cb->tx + pri;
        if (tx->nextp == NULL)
            tx->nextp = mb;
        else {
            for (mb2 = tx->nextp; mb2->m_act != NULL; mb2 = mb2->m_act)
                ;
            mb2->m_act = mb;
        }
```

Carlson   expires May 1998   [Page 22]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
    }


  4.2.3.  Frame Reception

    /*
     * This is called once for each FIFO-full-or-stale interrupt.  It
     * takes a pointer to the COBS state structure, the FIFO buffer
     * pointer, and the length of the data in the FIFO.  It in turn
     * calls send_up_packet(struct mbuf *) if a packet has been received,
     * or ppp_recv_error(void) if an error occurs, or disable_cobs(void)
     * if the peer is apparently restarting LCP negotiation.
     */
    void
    ppp_cobs_rcv(struct cobs_context *cb, u_char *buffer, int len)
    {
        u_char *dp = NULL,chr,rxlchar;
        struct mbuf *mb,*mb2;
        int mlen = 0,count;
        u_short fcs;

        if ((mb = cb->mtail) != NULL) {
            mlen = mb->m_len;
            dp = mtod(mb,u_char *) + mlen;
        }
        fcs = cb->rxfcs;
        count = cb->rxcount;
        rxlchar = chr = cb->rxlchar;
        while (len > 0) {
            len--;
            chr = *buffer++;
            if (chr == 0x7E) {
                rxlchar = chr;
                if (count > 0) {
                    if (mb != NULL)
                        mb->m_len = mlen;
                    m_freem(cb->savedpkt);
                    cb->savedfcs = fcs;
                    cb->savedpkt = cb->mhead;
                    cb->zadd = 0;
                    mb = cb->mhead = NULL;
                }
                while (cb->zadd > 1) {
                    if (mb == NULL || mlen >= MLEN) {
                        mb2 = m_get(M_DONTWAIT,MT_PPPRX);
                        if (mb2 == NULL)
                            goto ppp_input_error;
                        if (mb != NULL) {
```

Carlson   expires May 1998   [Page 23]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
                        mb->m_len = mlen;
                        mb->m_next = mb2;
                    } else
                        cb->mhead = mb2;
                    mb = mb2;
                    cb->mtail = mb;
                    dp = mtod(mb,u_char *);
                    mlen = 0;
                }
                *dp++ = 0;
                fcs = (fcs >> 8) ^ fcstab[fcs & 0xff];
                mlen++;
                cb->zadd--;
            }
            if (mb == NULL)
                m_freem(cb->mhead);
            else if (count > 0 || fcs != 0xF0B8) {
            ppp_input_error:
                if (cb->mhead->m_next != NULL || mlen > 4)
                    ppp_recv_error();
                m_freem(cb->mhead);
            } else {
                mb->m_len = mlen;
                m_adj(cb->mhead,-2);          /* Strip the CRC */
                send_up_packet(cb->mhead);
            }
            cb->mtail = cb->mhead = mb = NULL;
            count = cb->rxcount = cb->rxlcount = cb->zadd = 0;
            fcs = cb->rxfcs = 0xFFFF;
            continue;
        }
        if (chr == 0xD1 && rxlchar == 0x7E) {   /* Resume code */
            m_freem(cb->mhead);
            mb = cb->mhead = cb->savedpkt;
            if (mb != NULL) {
                while (mb->m_next != NULL)
                    mb = mb->m_next;
                cb->mtail = mb;
                mlen = mb->m_len;
                dp = mtod(mb,u_char *) + mlen;
            } else {
                dp = NULL;
                mlen = 0;
            }
            fcs = cb->savedfcs;
            cb->savedpkt = NULL;
            rxlchar = chr;
            continue;
```

  Carlson   expires May 1998   [Page 24]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
        }
        rxlchar = chr;
        if (chr == 0)
            chr = 0x7E;
        for (;;) {
            if (mb == NULL || mlen >= MLEN) {
                mb2 = m_get(M_DONTWAIT,MT_PPPRX);
                if (mb2 == NULL)
                    goto ppp_input_error;
                if (mb != NULL) {
                    mb->m_len = mlen;
                    mb->m_next = mb2;
                } else
                    cb->mhead = mb2;
                mb = mb2;
                cb->mtail = mb;
                dp = mtod(mb,u_char *);
                mlen = 0;
            }
            if (count > 0) {
                *dp++ = chr;
                fcs = (fcs >> 8) ^ fcstab[(fcs ^ chr) & 0xff];
                mlen++;
                count--;
                break;
            }
            if (cb->zadd == 0) {
        /* This can happen only if the peer starts renegotiating LCP */
                if (chr == 0xFF) {
                    disable_cobs();
                    m_freem(cb->mhead);
                    m_freem(cb->savedpkt);
                    cb->mhead = cb->mtail = cb->savedpkt = NULL;
                    return;
                }
                switch (chr & 0xF0) {
                case 0xD0:
                    cb->zadd = chr - 0xD0;
                    count = 0;
                    break;
                case 0xE0:
                case 0xF0:
                    count = chr - 0xE0;
                    cb->zadd = 2;
                    break;
                default:
                    cb->zadd = (chr == 0xD0) ? 0 : 1;
                    count = chr-1;
```

Carlson   expires May 1998   [Page 25]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

```
                }
                cb->rxlcount = count;
                break;
            }
            cb->zadd--;
            *dp++ = 0;
            fcs = (fcs >> 8) ^ fcstab[fcs & 0xff];
            mlen++;
        }
    }
    if (mb != NULL)
        mb->m_len = mlen;
    cb->rxfcs = fcs;
    cb->mtail = mb;
    cb->rxcount = count;
    cb->rxlchar = rxlchar;
}
```

5. Acknowledgments

This encapsulation method was first described in Stuart Cheshire's
Ph.D. Thesis at Stanford University.

6. References

[1] W. Simpson, "The Point-to-Point Protocol (PPP)", RFC 1661,
    07/21/1994

[2] W. Simpson, "PPP in HDLC-like Framing", RFC 1662, 07/1994

[3] S. Cheshire and M. Baker, "Consistent Overhead Byte Stuffing,"
    ACM SIGCOMM - Cannes, France, September 1997.

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

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

7. Authors' Addresses

James Carlson
IronBridge Networks
5 Corporate Drive

Carlson   expires May 1998   [Page 26]

INTERNET DRAFT   Consistent Overhead Byte Stuffing   November 1997

     Andover MA 01810-2448

     Phone: +1 978 691 4644
     Fax: +1 978 691 6300
     Email: carlson@wing.net

     Stuart Cheshire
     Stanford University
     Stanford CA 94305

     Phone: +1 650 723 9427
     Email: cheshire@cs.stanford.edu

     Mary Baker
     Stanford University
     Stanford CA 94305

     Phone: +1 650 725 3711
     Email: mgbaker@cs.stanford.edu

Carlson   expires May 1998   [Page 27]