                      PPP BSD Compression Protocol

Status of This Memo

   This memo provides information for the Internet community. This memo
   does not specify an Internet standard of any kind. Distribution of
   this memo is unlimited.

Abstract

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

   The PPP Compression Control Protocol [2] provides a method to
   negotiate and utilize compression protocols over PPP encapsulated
   links.

   This document describes the use of the Unix Compress compression
   protocol for compressing PPP encapsulated packets.

Table of Contents

1. Introduction

   UNIX compress as embodied in the freely and widely distributed BSD
   source has the following features:

      - dynamic table clearing when compression becomes less
        effective.

      - automatic turning off of compression when the overall result
        is not smaller than the input.

      - dynamic choice of code width within predetermined limits.

      - heavily used for many years in networks, on modem and other
        point-to-point links to transfer netnews.

      - an effective code width requires less than 64KBytes of memory
        on both sender and receive.

1.1. Licensing

   BSD Unix compress command source is widely and freely available, with
   no additional license for many computer vendors. The included source
   code is based on the BSD compress command source and carries only the
   copyright of The Regents of the University of California. Use the
   code entirely at your own risk. It has no warranties or
   indemnifications of any sort. Note that there are patents on LZW.

2. BSD Compress Packets

   Before any BSD Compress packets may be communicated, PPP must reach
   the Network-Layer Protocol phase, and the CCP Control Protocol must
   reach the Opened state.

   Exactly one BSD Compress datagram is encapsulated in the PPP
   Information field, where the PPP Protocol field contains 0xFD or
   0xFB. 0xFD is used when the PPP multilink protocol is not used or
   "above" multilink. 0xFB is used "below" multilink, to compress
   independently on individual links of a multilink bundle.

   The maximum length of the BSD Compress datagram transmitted over a
   PPP link is the same as the maximum length of the Information field
   of a PPP encapsulated packet.

   Only packets with PPP Protocol numbers in the range 0x0000 to 0x3FFF
   and neither 0xFD nor 0xFB are compressed. Other PPP packets are
   always sent uncompressed. Control packets are infrequent and should
   not be compressed for robustness.

   Padding

      BSD Compress packets require the previous negotiation of the
      Self-Describing-Padding Configuration Option [3] if padding is
      added to packets. If no padding is added, than Self-Describing-
      Padding is not required.

Reliability and Sequencing

   BSD Compress requires the packets to be delivered in sequence. It
   relies on Reset-Request and Reset-Ack CCP packets or on
   renegotiation of the Compression Control Protocol [2] to indicate
   loss of synchronization between the transmitter and receiver. The
   HDLC FCS detects corrupted packets and the normal mechanisms
   discard them. Missing or out of order packets are detected by the
   sequence number in each packet. The packet sequence number ought
   to be checked before decoding the packet.

   Instead of transmitting a Reset-Request packet when detecting a
   decompression error, the receiver MAY momentary force CCP to drop
   out of the Opened state by transmitting a new CCP Configure-
   Request. This method is more expensive than using Reset-Requests.

   When the receiver first encounters an unexpected sequence number
   it SHOULD send a Reset-Request CCP packet as defined in the
   Compression Control Protocol. When the transmitter sends the
   Reset-Ack or when the receiver receives a Reset-ACK, they must
   reset the sequence number to zero, clear the compression
   dictionary, and resume sending and receiving compressed packets.
   The receiver MUST discard all compressed packets after detecting
   an error and until it receives a Reset-Ack. This strategy can be
   thought of as abandoning the transmission of one "file" and
   starting the transmission of a new "file."

   The transmitter must clear its compression dictionary and respond
   with a Reset-Ack each time it receives a Reset-Request, because it
   cannot know if previous Reset-Acks reached the receiver. The
   receiver MUST clear its compression dictionary each time it
   receives a Reset-Ack, because the transmitter will have cleared
   its compression dictionary.

   When the link is busy, one decompression error is usually followed
   by several more before the Reset-Ack can be received. It is
   undesirable to transmit Reset-Requests more frequently than the
   round-trip-time of the link, because redundant Reset-Requests
   cause unnecessary compression dictionary clearing. The receiver
   MAY transmit an additional Reset-Request each time it receives a
   compressed or uncompressed packet until it finally receives a
   Reset-Ack, but the receiver ought not transmit another Reset-
   Request until the Reset-Ack for the previous one is late. The
   receiver MUST transmit enough Reset-Request packets to ensure that
   the transmitter receives at least one. For example, the receiver
   might choose to not transmit another Reset-Request until after one
   second (or, of course, a Reset-Ack has been received and
   decompression resumed).

   Data Expansion

      When significant data expansion is detected, the PPP packet MUST
      be sent without compression. Packets that would expand by fewer
      than 3 bytes SHOULD be sent without compression, but MAY be sent
      compressed provided the result does not exceed the MTU of the
      link. This makes moot standards document exegesises about exactly
      which bytes, such as the Protocol fields, count toward expansion.

      When a packet is received with PPP Protocol numbers in the range
      0x0000 to 0x3FFF, (except, of course, 0xFD and 0xFB) it is assumed
      that the packet would have caused expansion. The packet is
      locally compressed to update the compression history.

      Sending incompressible packets in their native encapsulation
      avoids maximum transmission unit complications. If uncompressed
      packets could be larger than their native form, then it would be
      necessary for the upper layers of an implementation to treat the
      PPP link as if it had a smaller MTU, to ensure that compressed
      incompressible packets are never larger than the negotiated PPP
   MTU.

      Using native encapsulation for incompressible packets complicates
      the implementation. The transmitter and the receiver must start
      putting information into the compression dictionary starting with
      the same packets, without relying upon seeing a compressed packet
      for synchronization. The first few packets after clearing the
      dictionary are usually incompressible, and so are likely to sent
      in their native encapsulation, just like packets before
      compression is turned on. If CCP or LCP packets are handled
      separately from Network-Layer packets (e.g. a "daemon" for control
      packets and "kernel code" for data packets), care must be taken to
      ensure that the transmitter synchronizes clearing the dictionary
      with the transmission of the configure-ACK or Reset-Ack that
      starts compression, and the receiver must similarly ensure that
      its dictionary is cleared before it processes the next packet.

      A difficulty caused by sending data that would expand uncompressed
      is that the receiver must adaptively clear its dictionary at
      precisely the same times as the sender. In the classic BSD
      compression code, the dictionary clearing is signaled by the
      reserved code 256. Because data that would expend is sent without
      compression, there is no reliable way for the sender to signal
      explicitly when it has cleared its dictionary. This difficulty is
      resolved by specifying the parameters that control the dictionary
      clearing, and having both sender and receiver clear their
      dictionaries at the same times.

2.1. Packet Format

   A summary of the BSD Compress packet format is shown below.

   The fields are transmitted from left to right.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           PPP Protocol          |             Sequence
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Data ...
   +-+-+-+-+-+-+-+-+
```
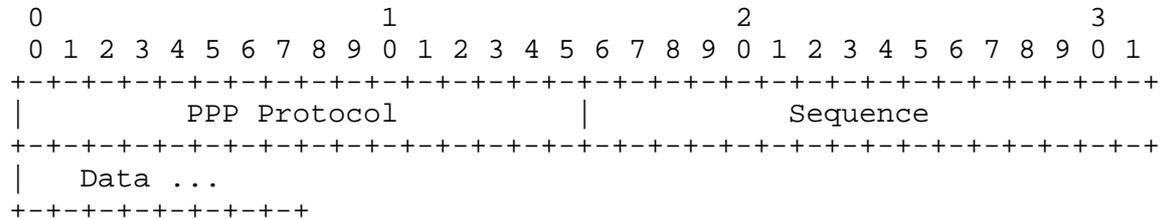
   PPP Protocol

      The PPP Protocol field is described in the Point-to-Point Protocol
      Encapsulation [1].

      When the BSD Compress compression protocol is successfully
      negotiated by the PPP Compression Control Protocol [2], the value
      of the protocol field is 0xFD or 0xFB. This value MAY be
      compressed when Protocol-Field-Compression is negotiated.

   Sequence

      The sequence number is sent most significant octet first. It
      starts at 0 when the dictionary is cleared, and is incremented by
      1 after each packet, including uncompressed packets. The sequence
      number after 65535 is zero. In other words, the sequence number
      "wraps" in the usual way.

      The sequence number ensures that lost or out of order packets do
      not cause the compression databases of the peers to become
      unsynchronized. When an unexpected sequence number is
      encountered, the dictionaries must be resynchronized with a CCP
      Reset-Request or Configure-Request. The packet sequence number
      can be checked before a compressed packet is decoded.

   Data

      The compressed PPP encapsulated packet, consisting of the Protocol
      and Data fields of the original, uncompressed packet follows.

      The Protocol field compression MUST be applied to the protocol
      field in the original packet before the sequence number is
      computed or the entire packet is compressed, regardless of whether

     the PPP protocol field compression has been negotiated. Thus, if
     the original protocol number was less than 0x100, it must be
     compressed to a single byte.

     The format of the compressed data is more precisely described by
     the example code in the "BSD Compress Algorithm" appendix.

3. Configuration Option Format

   Description

     The CCP BSD Compress Configuration Option negotiates the use of
     BSD Compress on the link. By default or ultimate disagreement, no
     compression is used.

   A summary of the BSD Compress Configuration Option format 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     | Vers|   Dict  |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
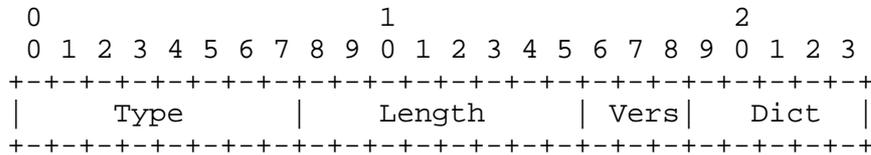
   Type

     21 or 0x15 for BSD compress.

   Length

     3

   Vers

     Must be the binary number 001.

   Dict

     The size in bits of the largest code used. It can range from 9 to
     16. A common choice is 12. The code included below can support
     code sizes from 9 to 15.

     It is convenient to treat the byte containing the Vers and Dict
     fields as a single field with legal values ranging from 0x29 to
     0x30.

     Note that the peer receiving compressed data must use the same
     code size as the peer sending data. It is not practical for the
     receiver to use a larger dictionary or code size, because both
     dictionaries must be cleared at the same time, even when the data
     is not compressible, so that uncompressed packets are being sent,
     and so the receiver cannot receive LZW "CLEAR" codes.

     When a received Configure-Request specifies a smaller dictionary
     than the local preference, it is often best to accept it instead
     of using a Configure-Nak to ask the peer to specify a larger
     dictionary.

A. BSD Compress Algorithm

   This code is the core of a commercial workstation implementation. It
   was derived by transliterating the 4.*BSD compress command. It is
   unlikely to be of direct use in any system that does not have the
   same mixture of mbufs and STREAMS buffers. It may need to be retuned
   for CPU's other than RISC's with many registers and certain
   addressing modes. However, the code is the most accurate and
   unambiguous way of defining the changes to the BSD compress source
   required to apply it to a stream instead of a file.

   Note that it assumes a "short" contains 16 bits and an "int" contains
   at least 32 bits. Where it would matter if more than 32 bits were in
   an "int" or "long," __uint32_t is used instead.

```
/* Because this code is derived from the 4.3BSD compress source:
 *
 *
 * Copyright (c) 1985, 1986 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * James A. Woods, derived from original work by Spencer Thomas
 * and Joseph Orost.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgement:
```

```
 *      This product includes software developed by the University of
 *      California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its
 *    contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


/* **************** */

struct bsd_db {
    int     totlen;                     /* length of this structure */
    u_int   hsize;                      /* size of the hash table */
    u_char  hshift;                     /* used in hash function */
    u_char  n_bits;                     /* current bits/code */
    u_char  debug;
    u_char  unit;
    u_short mru;
    u_short seqno;                      /* # of last byte of packet */
    u_int   maxmaxcode;                 /* largest valid code */
    u_int   max_ent;                    /* largest code in use */
    u_int   in_count;                   /* uncompressed bytes */
    u_int   bytes_out;                  /* compressed bytes */
    u_int   ratio;                      /* recent compression ratio */
    u_int   checkpoint;                 /* when to next check ratio */
    int     clear_count;                /* times dictionary cleared */
    int     incomp_count;               /* incompressible packets */
    int     decomp_count;               /* packets decompressed */
    int     overshoot;                  /* excess decompression buf */
    int     undershoot;                 /* insufficient decomp. buf */
    u_short *lens;                      /* array of lengths of codes */
    struct bsd_dict {
        union {                         /* hash value */
            __uint32_t  fcode;
            struct {
#ifdef BSD_LITTLE_ENDIAN
```

```
                u_short prefix;              /* preceding code */
                u_char  suffix;              /* last character of new code */
                u_char  pad;
#else
                u_char  pad;
                u_char  suffix;              /* last character of new code */
                u_short prefix;              /* preceding code */
#endif
            } hs;
        } f;
        u_short codem1;                      /* output of hash table -1 */
        u_short cptr;                        /* map code to hash table */
    } dict[1];
};
#define BSD_OVHD (2+2)                        /* overhead/packet */
#define MIN_BSD_BITS    9
#define MAX_BSD_BITS    15                    /* implementation limit */
#define BSD_VERS        1                     /* when shifted */
#ifdef _KERNEL
extern struct bsd_db *pf_bsd_init(struct bsd_db*, int, int, int);
extern int pf_bsd_comp(struct bsd_db*,u_char*,int,struct mbuf*,int);
extern mblk_t* pf_bsd_decomp(struct bsd_db*, mblk_t*);
extern void pf_bsd_incomp(struct bsd_db*, mblk_t*, u_int);
#endif


/* **************** */
/* PPP "BSD compress" compression
 *  The differences between this compression and the classic BSD LZW
 *  source are obvious from the requirement that the classic code worked
 *  with files while this handles arbitrarily long streams that
 *  are broken into packets.  They are:
 *
 *      When the code size expands, a block of junk is not emitted by
 *          the compressor and not expected by the decompressor.
 *
 *      New codes are not necessarily assigned every time an old
 *          code is output by the compressor.  This is because a packet
 *          end forces a code to be emitted, but does not imply that a
 *          new sequence has been seen.
 *
 *      The compression ratio is checked at the first end of a packet
 *          after the appropriate gap.  Besides simplifying and speeding
 *          things up, this makes it more likely that the transmitter
 *          and receiver will agree when the dictionary is cleared when
 *          compression is not going well.
 */
```

```
        /*
         * the next two codes should not be changed lightly, as they must not
         * lie within the contiguous general code space.
         */
        #define CLEAR   256                             /* table clear output code */
        #define FIRST   257                             /* first free entry */
        #define LAST    255

        #define BSD_INIT_BITS   MIN_BSD_BITS

        #define MAXCODE(b) ((1 << (b)) - 1)
        #define BADCODEM1 MAXCODE(MAX_BSD_BITS);

        #define BSD_HASH(prefix,suffix,hshift) ((((__uint32_t)(suffix)) \
                                                 << (hshift)) \
                                                ^ (__uint32_t)(prefix))
        #define BSD_KEY(prefix,suffix) ((((__uint32_t)(suffix)) << 16)  \
                                        + (__uint32_t)(prefix))

        #define CHECK_GAP       10000           /* Ratio check interval */

        #define RATIO_SCALE_LOG 8
        #define RATIO_SCALE     (1<<RATIO_SCALE_LOG)
        #define RATIO_MAX       (0x7fffffff>>RATIO_SCALE_LOG)

        /* clear the dictionary
         */
        static void
        pf_bsd_clear(struct bsd_db *db)
        {
                db->clear_count++;
                db->max_ent = FIRST-1;
                db->n_bits = BSD_INIT_BITS;
                db->ratio = 0;
                db->bytes_out = 0;
                db->in_count = 0;
                db->incomp_count = 0;
                db->decomp_count = 0;
                db->overshoot = 0;
                db->undershoot = 0;
                db->checkpoint = CHECK_GAP;
        }


        /* If the dictionary is full, then see if it is time to reset it.
         *
         * Compute the compression ratio using fixed-point arithmetic
         * with 8 fractional bits.
```

```
 *
 * Since we have an infinite stream instead of a single file,
 * watch only the local compression ratio.
 *
 * Since both peers must reset the dictionary at the same time even in
 * the absence of CLEAR codes (while packets are incompressible), they
 * must compute the same ratio.
 */
static int                              /* 1=output CLEAR */
pf_bsd_check(struct bsd_db *db)
{
        register u_int new_ratio;

        if (db->in_count >= db->checkpoint) {
                /* age the ratio by limiting the size of the counts */
                if (db->in_count >= RATIO_MAX
                    || db->bytes_out >= RATIO_MAX) {
                        db->in_count -= db->in_count/4;
                        db->bytes_out -= db->bytes_out/4;
                }

                db->checkpoint = db->in_count + CHECK_GAP;

                if (db->max_ent >= db->maxmaxcode) {
                        /* Reset the dictionary only if the ratio is
                         * worse, or if it looks as if it has been
                         * poisoned by incompressible data.
                         *
                         * This does not overflow, because
                         *      db->in_count <= RATIO_MAX.
                         */
                        new_ratio = db->in_count<<RATIO_SCALE_LOG;
                        if (db->bytes_out != 0)
                                new_ratio /= db->bytes_out;

                        if (new_ratio < db->ratio
                            || new_ratio < 1*RATIO_SCALE) {
                                pf_bsd_clear(db);
                                return 1;
                        }
                        db->ratio = new_ratio;
                }
        }
        return 0;
}


/* Initialize the database.
```

```
 */
struct bsd_db *
pf_bsd_init(struct bsd_db *db,          /* initialize this database */
            int unit,                   /* for debugging */
            int bits,                   /* size of LZW code word */
            int mru)                    /* MRU for input, 0 for output*/
{
        register int i;
        register u_short *lens;
        register u_int newlen, hsize, hshift, maxmaxcode;

        switch (bits) {
        case 9:                         /* needs 82152 for both comp &*/
        case 10:                        /* needs 84144          decomp*/
        case 11:                        /* needs 88240 */
        case 12:                        /* needs 96432 */
                hsize = 5003;
                hshift = 4;
                break;
        case 13:                        /* needs 176784 */
                hsize = 9001;
                hshift = 5;
                break;
        case 14:                        /* needs 353744 */
                hsize = 18013;
                hshift = 6;
                break;
        case 15:                        /* needs 691440 */
                hsize = 35023;
                hshift = 7;
                break;
        case 16:                        /* needs 1366160--far too much*/
                /* hsize = 69001; */    /* and 69001 is too big for */
                /* hshift = 8; */       /* cptr in struct bsd_db */
                /* break; */
        default:
                if (db) {
                        if (db->lens)
                                kern_free(db->lens);
                        kern_free(db);
                }
                return 0;
        }
        maxmaxcode = MAXCODE(bits);
        newlen = sizeof(*db) + (hsize-1)*(sizeof(db->dict[0]));

        if (db) {
                lens = db->lens;
```

```
                if (db->totlen != newlen) {
                        if (lens)
                                kern_free(lens);
                        kern_free(db);
                        db = 0;
                }
        }
        if (!db) {
                db = (struct bsd_db*)kern_malloc(newlen);
                if (!db)
                        return 0;
                if (mru == 0) {
                        lens = 0;
                } else {
                        lens = (u_short*)kern_malloc((maxmaxcode+1)
                                                * sizeof(*lens));
                        if (!lens) {
                                kern_free(db);
                                return 0;
                        }
                        i = LAST+1;
                        while (i != 0)
                                lens[--i] = 1;
                }
                i = hsize;
                while (i != 0) {
                        db->dict[--i].codem1 = BADCODEM1;
                        db->dict[i].cptr = 0;
                }
        }

        bzero(db,sizeof(*db)-sizeof(db->dict));
        db->lens = lens;
        db->unit = unit;
        db->mru = mru;
        db->hsize = hsize;
        db->hshift = hshift;
        db->maxmaxcode = maxmaxcode;
        db->clear_count = -1;

        pf_bsd_clear(db);

        return db;
}


/* compress a packet
 *      Assume the protocol is known to be >= 0x21 and < 0xff.
```

```
 *         One change from the BSD compress command is that when the
 *         code size expands, we do not output a bunch of padding.
 */
int                                       /* new slen */
pf_bsd_comp(struct bsd_db *db,
            u_char *cp_buf,               /* compress into here */
            int proto,                    /* this original PPP protocol */
            struct mbuf *m,               /* from here */
            int slen)
{
        register int hshift = db->hshift;
        register u_int max_ent = db->max_ent;
        register u_int n_bits = db->n_bits;
        register u_int bitno = 32;
        register __uint32_t accum = 0;
        register struct bsd_dict *dictp;
        register __uint32_t fcode;
        register u_char c;
        register int hval, disp, ent;
        register u_char *rptr, *wptr;
        struct mbuf *n;

#define OUTPUT(ent) {                     \
        bitno -= n_bits;                  \
        accum |= ((ent) << bitno);        \
        do {                              \
                *wptr++ = accum>>24;      \
                accum <<= 8;              \
                bitno += 8;               \
        } while (bitno <= 24);            \
        }

        /* start with the protocol byte */
        ent = proto;
        db->in_count++;

        /* install sequence number */
        cp_buf[0] = db->seqno>>8;
        cp_buf[1] = db->seqno;
        db->seqno++;

        wptr = &cp_buf[2];
        slen = m->m_len;
        db->in_count += slen;
        rptr = mtod(m, u_char*);
        n = m->m_next;
        for (;;) {
```

```
                if (slen == 0) {
                        if (!n)
                                break;
                        slen = n->m_len;
                        rptr = mtod(n, u_char*);
                        n = n->m_next;
                        if (!slen)
                                continue;   /* handle 0-length buffers*/
                        db->in_count += slen;
                }

                slen--;
                c = *rptr++;
                fcode = BSD_KEY(ent,c);
                hval = BSD_HASH(ent,c,hshift);
                dictp = &db->dict[hval];

                /* Validate and then check the entry. */
                if (dictp->codem1 >= max_ent)
                        goto nomatch;
                if (dictp->f.fcode == fcode) {
                        ent = dictp->codem1+1;
                        continue;           /* found (prefix,suffix) */
                }

                /* continue probing until a match or invalid entry */
                disp = (hval == 0) ? 1 : hval;
                do {
                        hval += disp;
                        if (hval >= db->hsize)
                                hval -= db->hsize;
                        dictp = &db->dict[hval];
                        if (dictp->codem1 >= max_ent)
                                goto nomatch;
                } while (dictp->f.fcode != fcode);
                ent = dictp->codem1+1;  /* found (prefix,suffix) */
                continue;

        nomatch:
                OUTPUT(ent);                  /* output the prefix */

                /* code -> hashtable */
                if (max_ent < db->maxmaxcode) {
                        struct bsd_dict *dictp2;
                        /* expand code size if needed */
                        if (max_ent >= MAXCODE(n_bits))
                                db->n_bits = ++n_bits;
```

```
                                /* Invalidate old hash table entry using
                                 * this code, and then take it over.
                                 */
                                dictp2 = &db->dict[max_ent+1];
                                if (db->dict[dictp2->cptr].codem1 == max_ent)
                                        db->dict[dictp2->cptr].codem1=BADCODEM1;
                                dictp2->cptr = hval;
                                dictp->codem1 = max_ent;
                                dictp->f.fcode = fcode;

                                db->max_ent = ++max_ent;
                        }
                        ent = c;
                }

        OUTPUT(ent);                            /* output the last code */
        db->bytes_out += (wptr-&cp_buf[2]   /* count complete bytes */
                            + (32-bitno+7)/8);

        if (pf_bsd_check(db))
                OUTPUT(CLEAR);              /* do not count the CLEAR */

        /* Pad dribble bits of last code with ones.
         * Do not emit a completely useless byte of ones.
         */
        if (bitno != 32)
                *wptr++ = (accum | (0xff << (bitno-8))) >> 24;

        /* Increase code size if we would have without the packet
         * boundary and as the decompressor will.
         */
        if (max_ent >= MAXCODE(n_bits)
            && max_ent < db->maxmaxcode)
                db->n_bits++;

        return (wptr - cp_buf);
#undef OUTPUT
}


/* Update the "BSD Compress" dictionary on the receiver for
 * incompressible data by pretending to compress the incoming data.
 */
void
pf_bsd_incomp(struct bsd_db *db,
                mblk_t *dmsg,
                u_int ent)                      /* start with protocol byte */
{
```

```
        register u_int hshift = db->hshift;
        register u_int max_ent = db->max_ent;
        register u_int n_bits = db->n_bits;
        register struct bsd_dict *dictp;
        register __uint32_t fcode;
        register u_char c;
        register int hval, disp;
        register int slen;
        register u_int bitno = 7;
        register u_char *rptr;

        db->incomp_count++;

        db->in_count++;                     /* count protocol as 1 byte */
        db->seqno++;
        rptr = dmsg->b_rptr+PPP_BUF_HEAD_INFO;
        for (;;) {
                slen = dmsg->b_wptr - rptr;
                if (slen == 0) {
                        dmsg = dmsg->b_cont;
                        if (!dmsg)
                                break;
                        rptr = dmsg->b_rptr;
                        continue;           /* skip zero-length buffers */
                }
                db->in_count += slen;

                do {
                        c = *rptr++;
                        fcode = BSD_KEY(ent,c);
                        hval = BSD_HASH(ent,c,hshift);
                        dictp = &db->dict[hval];

                        /* validate and then check the entry */
                        if (dictp->codem1 >= max_ent)
                                goto nomatch;
                        if (dictp->f.fcode == fcode) {
                                ent = dictp->codem1+1;
                                continue;    /* found (prefix,suffix) */
                        }

                        /* continue until match or invalid entry */
                        disp = (hval == 0) ? 1 : hval;
                        do {
                                hval += disp;
                                if (hval >= db->hsize)
                                        hval -= db->hsize;
                                dictp = &db->dict[hval];
```

```
                                 if (dictp->codem1 >= max_ent)
                                         goto nomatch;
                         } while (dictp->f.fcode != fcode);
                         ent = dictp->codem1+1;
                         continue;        /* found (prefix,suffix) */

nomatch:                                  /* output (count) the prefix */
                         bitno += n_bits;

                         /* code -> hashtable */
                         if (max_ent < db->maxmaxcode) {
                             struct bsd_dict *dictp2;
                             /* expand code size if needed */
                             if (max_ent >= MAXCODE(n_bits))
                                     db->n_bits = ++n_bits;
                             /* Invalidate previous hash table entry
                              * assigned this code, and then take it over
                              */
                             dictp2 = &db->dict[max_ent+1];
                             if (db->dict[dictp2->cptr].codem1==max_ent)
                                db->dict[dictp2->cptr].codem1=BADCODEM1;
                             dictp2->cptr = hval;
                             dictp->codem1 = max_ent;
                             dictp->f.fcode = fcode;

                             db->max_ent = ++max_ent;
                             db->lens[max_ent] = db->lens[ent]+1;
                         }
                         ent = c;
                 } while (--slen != 0);
        }
        bitno += n_bits;                    /* output (count) last code */
        db->bytes_out += bitno/8;

        (void)pf_bsd_check(db);

        /* Increase code size if we would have without the packet
         * boundary and as the decompressor will.
         */
        if (max_ent >= MAXCODE(n_bits)
            && max_ent < db->maxmaxcode)
                db->n_bits++;
}


/* Decompress "BSD Compress"
 */
mblk_t*                                   /* 0=failed, so zap CCP */
```

```
pf_bsd_decomp(struct bsd_db *db,
            mblk_t *cmsg)
{
        register u_int max_ent = db->max_ent;
        register __uint32_t accum = 0;
        register u_int bitno = 32;        /* 1st valid bit in accum */
        register u_int n_bits = db->n_bits;
        register u_int tgtbitno = 32-n_bits; /* bitno when accum full */
        register struct bsd_dict *dictp;
        register int explen, i;
        register u_int incode, oldcode, finchar;
        register u_char *p, *rptr, *rptr9, *wptr0, *wptr;
        mblk_t *dmsg, *dmsg1, *bp;

        db->decomp_count++;
        rptr = cmsg->b_rptr;
        ASSERT(cmsg->b_wptr >= rptr+PPP_BUF_MIN);
        ASSERT(PPP_BUF_ALIGN(rptr));
        rptr += PPP_BUF_MIN;

        /* get the sequence number */
        i = 0;
        explen = 2;
        do {
                while (rptr >= cmsg->b_wptr) {
                        bp = cmsg;
                        cmsg = cmsg->b_cont;
                        freeb(bp);
                        if (!cmsg) {
                                if (db->debug)
                                        printf("bsd_decomp%d: missing"
                                                " %d header bytes\n",
                                                db->unit, explen);
                                return 0;
                        }
                        rptr = cmsg->b_rptr;
                }
                i = (i << 8) + *rptr++;
        } while (--explen != 0);
        if (i != db->seqno++) {
                freemsg(cmsg);
                if (db->debug)
                        printf("bsd_decomp%d: bad sequence number 0x%x"
                                " instead of 0x%x\n",
                                db->unit, i, db->seqno-1);
                return 0;
        }
```

```
           /* Guess how much memory we will need.  Assume this packet was
            * compressed by at least 1.5X regardless of the recent ratio.
            */
           if (db->ratio > (RATIO_SCALE*3)/2)
                   explen = (msgdsize(cmsg)*db->ratio)/RATIO_SCALE;
           else
                   explen = (msgdsize(cmsg)*3)/2;
           if (explen > db->mru)
                   explen = db->mru;

           dmsg = dmsg1 = allocb(explen+PPP_BUF_HEAD_INFO, BPRI_HI);
           if (!dmsg1) {
                   freemsg(cmsg);
                   return 0;
           }

           wptr = dmsg1->b_wptr;

           ((struct ppp_buf*)wptr)->type = BEEP_FRAME;
           /* the protocol field must be compressed */
           ((struct ppp_buf*)wptr)->proto = 0;
           wptr += PPP_BUF_HEAD_PROTO+1;

           rptr9 = cmsg->b_wptr;
           db->bytes_out += rptr9-rptr;
           wptr0 = wptr;
           explen = dmsg1->b_datap->db_lim - wptr;
           oldcode = CLEAR;
           for (;;) {
                   if (rptr >= rptr9) {
                           bp = cmsg;
                           cmsg = cmsg->b_cont;
                           freeb(bp);
                           if (!cmsg)      /* quit at end of message */
                                   break;
                           rptr = cmsg->b_rptr;
                           rptr9 = cmsg->b_wptr;
                           db->bytes_out += rptr9-rptr;
                           continue;       /* handle 0-length buffers */
                   }

                   /* Accumulate bytes until we have a complete code.
                    * Then get the next code, relying on the 32-bit,
                    * unsigned accum to mask the result.
                    */
                   bitno -= 8;
                   accum |= *rptr++ << bitno;
                   if (tgtbitno < bitno)
```

```
                                continue;
                        incode = accum >> tgtbitno;
                        accum <<= n_bits;
                        bitno += n_bits;

                        if (incode == CLEAR) {
                                /* The dictionary must only be cleared at
                                 * the end of a packet.  But there could be an
                                 * empty message block at the end.
                                 */
                                if (rptr != rptr9
                                    || cmsg->b_cont != 0) {
                                        cmsg->b_rptr = rptr;
                                        i = msgdsize(cmsg);
                                        if (i != 0) {
                                                freemsg(dmsg);
                                                freemsg(cmsg);
                                                if (db->debug)
                                                        printf("bsd_decomp%d: "
                                                                "bad CLEAR\n",
                                                                db->unit);
                                                return 0;
                                        }
                                }
                                pf_bsd_clear(db);
                                freemsg(cmsg);
                                wptr0 = wptr;
                                break;
                        }

                        /* Special case for KwKwK string. */
                        if (incode > max_ent) {
                                if (incode > max_ent+2
                                    || incode > db->maxmaxcode
                                    || oldcode == CLEAR) {
                                        freemsg(dmsg);
                                        freemsg(cmsg);
                                        if (db->debug)
                                            printf("bsd_decomp%d: bad code %x\n",
                                                    db->unit, incode);
                                        return 0;
                                }
                                i = db->lens[oldcode];
                                /* do not write past end of buf */
                                explen -= i+1;
                                if (explen < 0) {
                                        db->undershoot -= explen;
                                        db->in_count += wptr-wptr0;
```

```
                            dmsg1->b_wptr = wptr;
                            CK_WPTR(dmsg1);
                            explen = MAX(64,i+1);
                            bp = allocb(explen, BPRI_HI);
                            if (!bp) {
                                    freemsg(cmsg);
                                    freemsg(dmsg);
                                    return 0;
                            }
                            dmsg1->b_cont = bp;
                            dmsg1 = bp;
                            wptr0 = wptr = dmsg1->b_wptr;
                          explen=dmsg1->b_datap->db_lim-wptr-(i+1);
                    }
                    p = (wptr += i);
                    *wptr++ = finchar;
                    finchar = oldcode;
            } else {
                    i = db->lens[finchar = incode];
                    explen -= i;
                    if (explen < 0) {
                            db->undershoot -= explen;
                            db->in_count += wptr-wptr0;
                            dmsg1->b_wptr = wptr;
                            CK_WPTR(dmsg1);
                            explen = MAX(64,i);
                            bp = allocb(explen, BPRI_HI);
                            if (!bp) {
                                    freemsg(dmsg);
                                    freemsg(cmsg);
                                    return 0;
                            }
                            dmsg1->b_cont = bp;
                            dmsg1 = bp;
                            wptr0 = wptr = dmsg1->b_wptr;
                            explen = dmsg1->b_datap->db_lim-wptr-i;
                    }
                    p = (wptr += i);
            }

            /* decode code and install in decompressed buffer */
            while (finchar > LAST) {
                    dictp = &db->dict[db->dict[finchar].cptr];
                    *--p = dictp->f.hs.suffix;
                    finchar = dictp->f.hs.prefix;
            }
            *--p = finchar;
```

```
                     /* If not first code in a packet, and
                      * if not out of code space, then allocate a new code.
                      *
                      * Keep the hash table correct so it can be used
                      * with uncompressed packets.
                      */
                    if (oldcode != CLEAR
                        && max_ent < db->maxmaxcode) {
                            struct bsd_dict *dictp2;
                            __uint32_t fcode;
                            int hval, disp;

                            fcode = BSD_KEY(oldcode,finchar);
                            hval = BSD_HASH(oldcode,finchar,db->hshift);
                            dictp = &db->dict[hval];
                            /* look for a free hash table entry */
                            if (dictp->codem1 < max_ent) {
                                    disp = (hval == 0) ? 1 : hval;
                                    do {
                                            hval += disp;
                                            if (hval >= db->hsize)
                                                    hval -= db->hsize;
                                            dictp = &db->dict[hval];
                                    } while (dictp->codem1 < max_ent);
                            }

                            /* Invalidate previous hash table entry
                             * assigned this code, and then take it over
                             */
                            dictp2 = &db->dict[max_ent+1];
                            if (db->dict[dictp2->cptr].codem1 == max_ent) {
                                    db->dict[dictp2->cptr].codem1=BADCODEM1;
                            }
                            dictp2->cptr = hval;
                            dictp->codem1 = max_ent;
                            dictp->f.fcode = fcode;

                            db->max_ent = ++max_ent;
                            db->lens[max_ent] = db->lens[oldcode]+1;

                            /* Expand code size if needed.
                             */
                            if (max_ent >= MAXCODE(n_bits)
                                && max_ent < db->maxmaxcode) {
                                    db->n_bits = ++n_bits;
                                    tgtbitno = 32-n_bits;
                            }
                    }
```

```
                  oldcode = incode;
        }

        db->in_count += wptr-wptr0;
        dmsg1->b_wptr = wptr;
        CK_WPTR(dmsg1);

        db->overshoot += explen;

        /* Keep the checkpoint right so that incompressible packets
         * clear the dictionary at the right times.
         */
        if (pf_bsd_check(db)
            && db->debug) {
              printf("bsd_decomp%d: peer should have "
                      "cleared dictionary\n", db->unit);
        }

        return dmsg;
}
```

Security Considerations

    Security issues are not discussed in this memo.

References

    [1]   Simpson, W., "The Point-to-Point Protocol (PPP)", STD 51,
           RFC 1661, July 1994.

    [2]   Rand, D., "The PPP Compression Control Protocol (CCP)", RFC
           1962, June 1996.

    [3]   Simpson, W., "PPP LCP Extensions", RFC 1570, January 1994.

    [4]   Simpson, W., "PPP in HDLC-like Framing", STD 51, RFC 1662,
           July 1994.

Acknowledgments

    William Simpson provided and supported the very valuable idea of not
    using any additional header bytes for incompressible packets.

Chair's Address

    The working group can be contacted via the current chair:

    Karl Fox
    Ascend Communications
    3518 Riverside Drive, Suite 101
    Columbus, Ohio 43221

    EMail: karl@ascend.com

Author's Address

    Questions about this memo can also be directed to:

    Vernon Schryver
    2482 Lee Hill Drive
    Boulder, Colorado 80302

    EMail: vjs@rhyolite.com