A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP

INTRODUCTION

The TCP/IP protocol family runs over a variety of network media: IEEE 802.3 (ethernet) and 802.5 (token ring) LAN's, X.25 lines, satellite links, and serial lines. There are standard encapsulations for IP packets defined for many of these networks, but there is no standard for serial lines. SLIP, Serial Line IP, is a currently a de facto standard, commonly used for point-to-point serial connections running TCP/IP. It is not an Internet standard. Distribution of this memo is unlimited.

HISTORY

SLIP has its origins in the 3COM UNET TCP/IP implementation from the early 1980's. It is merely a packet framing protocol: SLIP defines a sequence of characters that frame IP packets on a serial line, and nothing more. It provides no addressing, packet type identification, error detection/correction or compression mechanisms. Because the protocol does so little, though, it is usually very easy to implement.

Around 1984, Rick Adams implemented SLIP for 4.2 Berkeley Unix and Sun Microsystems workstations and released it to the world. It quickly caught on as an easy reliable way to connect TCP/IP hosts and routers with serial lines.

SLIP is commonly used on dedicated serial links and sometimes for dialup purposes, and is usually used with line speeds between 1200bps and 19.2Kbps. It is useful for allowing mixes of hosts and routers to communicate with one another (host-host, host-router and router-router are all common SLIP network configurations).

AVAILABILITY

SLIP is available for most Berkeley UNIX-based systems. It is included in the standard 4.3BSD release from Berkeley. SLIP is available for Ultrix, Sun UNIX and most other Berkeley-derived UNIX systems. Some terminal concentrators and IBM PC implementations also support it.

SLIP for Berkeley UNIX is available via anonymous FTP from uunet.uu.net in pub/sl.shar.Z. Be sure to transfer the file in binary mode and then run it through the UNIX uncompress program. Take

the resulting file and use it as a shell script for the UNIX /bin/sh (for instance, /bin/sh sl.shar).

PROTOCOL

The SLIP protocol defines two special characters: END and ESC. END is octal 300 (decimal 192) and ESC is octal 333 (decimal 219) not to be confused with the ASCII ESCape character; for the purposes of this discussion, ESC will indicate the SLIP ESC character. To send a packet, a SLIP host simply starts sending the data in the packet. If a data byte is the same code as END character, a two byte sequence of ESC and octal 334 (decimal 220) is sent instead. If it the same as an ESC character, an two byte sequence of ESC and octal 335 (decimal 221) is sent instead. When the last byte in the packet has been sent, an END character is then transmitted.

Phil Karn suggests a simple change to the algorithm, which is to begin as well as end packets with an END character. This will flush any erroneous bytes which have been caused by line noise. In the normal case, the receiver will simply see two back-to-back END characters, which will generate a bad IP packet. If the SLIP implementation does not throw away the zero-length IP packet, the IP implementation certainly will. If there was line noise, the data received due to it will be discarded without affecting the following packet.

Because there is no ’standard’ SLIP specification, there is no real defined maximum packet size for SLIP. It is probably best to accept the maximum packet size used by the Berkeley UNIX SLIP drivers: 1006 bytes including the IP and transport protocol headers (not including the framing characters). Therefore any new SLIP implementations should be prepared to accept 1006 byte datagrams and should not send more than 1006 bytes in a datagram.

DEFICIENCIES

There are several features that many users would like SLIP to provide which it doesn’t. In all fairness, SLIP is just a very simple protocol designed quite a long time ago when these problems were not really important issues. The following are commonly perceived shortcomings in the existing SLIP protocol:

- addressing:

  both computers in a SLIP link need to know each other’s IP addresses for routing purposes. Also, when using SLIP for hosts to dial-up a router, the addressing scheme may be quite dynamic and the router may need to inform the dialing host of

the host's IP address. SLIP currently provides no mechanism for hosts to communicate addressing information over a SLIP connection.

- type identification:

  SLIP has no type field. Thus, only one protocol can be run over a SLIP connection, so in a configuration of two DEC computers running both TCP/IP and DECnet, there is no hope of having TCP/IP and DECnet share one serial line between them while using SLIP. While SLIP is "Serial Line IP", if a serial line connects two multi-protocol computers, those computers should be able to use more than one protocol over the line.

- error detection/correction:

  noisy phone lines will corrupt packets in transit. Because the line speed is probably quite low (likely 2400 baud), retransmitting a packet is very expensive. Error detection is not absolutely necessary at the SLIP level because any IP application should detect damaged packets (IP header and UDP and TCP checksums should suffice), although some common applications like NFS usually ignore the checksum and depend on the network media to detect damaged packets. Because it takes so long to retransmit a packet which was corrupted by line noise, it would be efficient if SLIP could provide some sort of simple error correction mechanism of its own.

- compression:

  because dial-in lines are so slow (usually 2400bps), packet compression would cause large improvements in packet throughput. Usually, streams of packets in a single TCP connection have few changed fields in the IP and TCP headers, so a simple compression algorithms might just send the changed parts of the headers instead of the complete headers.

Some work is being done by various groups to design and implement a successor to SLIP which will address some or all of these problems.

SLIP DRIVERS

The following C language functions send and receive SLIP packets. They depend on two functions, send_char() and recv_char(), which send and receive a single character over the serial line.

```
/* SLIP special character codes
 */
#define END             0300    /* indicates end of packet */
#define ESC             0333    /* indicates byte stuffing */
#define ESC_END         0334    /* ESC ESC_END means END data byte */
#define ESC_ESC         0335    /* ESC ESC_ESC means ESC data byte */

/* SEND_PACKET: sends a packet of length "len", starting at
 * location "p".
 */
void send_packet(p, len)
        char *p;
        int len; {

  /* send an initial END character to flush out any data that may
   * have accumulated in the receiver due to line noise
   */
     send_char(END);

  /* for each byte in the packet, send the appropriate character
   * sequence
   */
        while(len--) {
                switch(*p) {
                /* if it's the same code as an END character, we send a
                 * special two character code so as not to make the
                 * receiver think we sent an END
                 */
                case END:
                        send_char(ESC);
                        send_char(ESC_END);
                        break;

                /* if it's the same code as an ESC character,
                 * we send a special two character code so as not
                 * to make the receiver think we sent an ESC
                 */
                case ESC:
                        send_char(ESC);
                        send_char(ESC_ESC);
                        break;
```

```
                /* otherwise, we just send the character
                 */
                default:
                        send_char(*p);
                        }

                p++;
                }

        /* tell the receiver that we're done sending the packet
         */
        send_char(END);
        }

/* RECV_PACKET: receives a packet into the buffer located at "p".
 *      If more than len bytes are received, the packet will
 *      be truncated.
 *      Returns the number of bytes stored in the buffer.
 */
int recv_packet(p, len)
        char *p;
        int len; {
        char c;
        int received = 0;

        /* sit in a loop reading bytes until we put together
         * a whole packet.
         * Make sure not to copy them into the packet if we
         * run out of room.
         */
        while(1) {
                /* get a character to process
                 */
                c = recv_char();

                /* handle bytestuffing if necessary
                 */
                switch(c) {

                /* if it's an END character then we're done with
                 * the packet
                 */
                case END:
                        /* a minor optimization: if there is no
                         * data in the packet, ignore it. This is
                         * meant to avoid bothering IP with all
                         * the empty packets generated by the
                         * duplicate END characters which are in
```

```
                         * turn sent to try to detect line noise.
                         */
                        if(received)
                                return received;
                        else
                                break;

                /* if it's the same code as an ESC character, wait
                 * and get another character and then figure out
                 * what to store in the packet based on that.
                 */
                case ESC:
                        c = recv_char();

                        /* if "c" is not one of these two, then we
                         * have a protocol violation.  The best bet
                         * seems to be to leave the byte alone and
                         * just stuff it into the packet
                         */
                        switch(c) {
                        case ESC_END:
                                c = END;
                                break;
                        case ESC_ESC:
                                c = ESC;
                                break;
                                }

                /* here we fall into the default handler and let
                 * it store the character for us
                 */
                default:
                        if(received < len)
                                p[received++] = c;
                        }
                }
        }
```