Adobe SCSI Input Protocol Specification

Adobe Developer Support

Version 1.0

18 March 1993

Adobe Systems Incorporated

Corporate Headquarters
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400 Main Number
(415) 961-4111 Developer Support
Fax: (415) 961-3769

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z-O, Netherlands
+31-20-6511 200
Fax: +31-20-6511 300

Adobe Systems Japan
Swiss Bank House 7F
4-1-8 Toranomon, Minato-ku
Tokyo 105, Japan
+81-3-3437-8950
Fax: +81-3-3437-8968

PN LPS5010

PostScript is a trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Any references to a "PostScript printer," a "PostScript file," or a "PostScript driver" refer to printers, files, and driver programs (respectively) which are written in or support the PostScript language. The sentences in this book that use "PostScript language" as an adjective phrase are so constructed to reinforce that the name refers to the standard language definition as set forth by Adobe Systems Incorporated.

Contents

Adobe™ SCSI Input Protocol Specification

1 Introduction

This document specifies the SCSI communications protocol for transmitting data from a host to a PostScript™ language interpreter. Though the data being transmitted are typically PostScript language programs nothing in the protocol requires this. This is a partial specification describing the currently implemented protocols. Other protocols will be added to this document as they are implemented

The target audience for this document are software developers implementing the host-side SCSI communication driver. Extensive knowledge of the Small Computer System Interface specification (X3.131, alias SCSI) is assumed. This document is meant to be used with ANSI X3.131. For more information about SCSI interfaces in general please refer to the ANSI specification.

In general the protocol can be simply described as a device to device communication where one device initiates the transfer of information to the other by specifying the amount of data that it is to be sent. This communication is described as *initiator* to *target* in the ANSI specification for Small Computer System Interface. In this specification it will be referred to as host to RIP (Raster Image Processor) communication. The host will send a write command with the count of the number of bytes to the RIP. The RIP then reads this number of bytes until it reaches the end of the transmission or is interrupted. In the case of an interrupt the RIP requests status information from the host. If the RIP reads until the end of transmission it acknowledges the EOT and waits for the next job stream.

Data is transferred on one of two data streams the in-band data stream or the out-of-band data stream. Data sent on the in-band data stream is called in-band data and is the driving force behind the transmission. When status information needs to be conveyed between host and RIP it is sent via the out-of-band data stream and is called out-of-band data. Each data stream has its own

protocol but there is a standard packet format used to convey information for both. The packet structure and the protocols for using the in-band and out-of-band data streams are described in this specification.

In General

Some constants of the implementation are defined here and have the same meaning throughout this document.

- All numbers preceded by a *0x* are in hexadecimal notation.
- A sector is defined as 512 bytes.
- Sector counts in the SCSI command block indicate the number of 512 byte blocks.
- A sector address is the *Logical Block Addres*s in the SCSI command block.
- Unless otherwise stated the protocol descriptions will be the same for both a Level 1 PostScript Interpreter and a Level 2 PostScript Interpreter.

The example code in this document was written for a Unix® host computer. Other hosts will have different implementations. Refer to Appendix A for a complete listing of the sample application for a Unix host.

2 SCSI Protocol

There are six phases used to convey information over a SCSI channel. These are the Bus Free, Selection, Message, Command, Data, and Status phases. These phases provide the basis for all communication over the SCSI channel. The Bus Free and Selection phases determine when the channel is available and how to select it. The Message phase provides a mechanism for physical path management. The Command, Data, and Status phases convey commands as well as the physical data and status information to and from the host and RIP. The communication of data or status information is typically known as the in-band and out-of-band data stream respectively.

2.1 Bus Free and Selection Phases

The *Bus Free* phase and *Selection* phases are used to free the bus for use and to select a given device on the bus. The *Bus Free* phase begins after all transmissions on the channel are complete and the bus is idle. The *Selection* phase begins during the *Bus Free* phase and allows the host to select a device to initiate a Read or Write.

2.2 Message Phase

The Level 2 interpreter processes *Messages Out* that are requested and sent by the host. The RIP will never generate a *Message Out* except in response to an ATTENTION raised from the host. An example of when this phase is invoked would be after the host has raised the ATTENTION condition to at the beginning of a transaction to identify the RIP. The RIP will send only the following message codes:

Table 1 *Messages Out*

Description	*Code*
Message Reject	0x7
No Operation	0x8
Identify	0x80 - 0xf8 (the least 3 significant bits, which specify a logical unit number in a RIP, are expected to be all 0)

With all other message codes, the RIP will send a Message Reject message to the host. When all the message bytes have been received successfully the RIP will indicate its success by changing to either the Command, Data or Status phases. In level 1, the RIP does not process Messages Out and will never respond to the host's ATTENTION with the *Messages Out* phase.

2.3 Command, Data and Status Phases

This is the phase that conveys information between devices. The type of information is determined by the state of the transmission and the data stream used. The following SCSI standard commands are currently implemented unless otherwise specified:

Table 2 *Implemented SCSI commands*

Command	*Opcode*
Test unit ready	0x00
Rezero unit	0x01
Request sense	0x03
Format Unit	0x04*
Read	0x08
Write	0x0a
Seek	0x0b
Inquiry	0x12

Reserve Unit	0x16*
Release Unit	0x17*
Start/Stop Unit	0x1b*
Send Diagnostic	0x1d*
Read capacity	0x25
Read Extended	0x28*
Write Extended	0x2a*

*Note: * Indicates a command supported only in Level 2*

The details of how these commands interact with the PostScript interpreter will be presented below.

2.4 Test Unit Ready

Test unit ready will return a *GOOD* status once the PostScript interpreter is booted with the SCSI stream enabled (All Sys/Start and/or Sys/Bootlist processing is complete). Note the interpreter itself may not be idle at this point. All other conditions will return *CHECK CONDITION* status.

2.5 Rezero Unit

Rezero unit will be ignored and will always return *GOOD* status.

2.6 Request Sense

Request sense command requests data from the RIP about the previous command after the host receives a CHECK CONDITION status. Request Sense will return a CHECK CONDITION only to report fatal errors at the RIP. All non-fatal errors will return a GOOD status. Sense data is preserved by the RIP until a *Request Sense* command is received or another command is received from the same host.

Level 1

Request sense for the Level 1 interpreter will return only the non-extended sense data format for error classes 0 through 6. The vendor unique field will always be zero. *GOOD* status will always be returned. Error codes are:

Table 3 *Error Codes*

Code	*Meaning*
0x00	Unspecified error
0x01	Illegal Logical Sector
0x02	Sector Transfer Out of Range
0x05	E_Invalid_address
0x25	K_Illegal_Request

Non-extended sense data is supported in Level 1 only.

Level 2

Request sense data for Level 2 will be formatted in the extended mode only. It will accept an error class of 7, which specifies extended sense, and an error code of 0, which denotes the e extended sense data format. The sense key returned will be from the following table:

Table 4 *Sense Keys*

Description	*Code*
No Sense	0x00
Hardware Error	0x04
Illegal Request	0x05

Sense keys provide information about the type of data being returned from the RIP. Additional information about the current CHECK CONDITION is held in the Information Byte fields of the sense data packet. If more than 12 bytes of data are requested in the *Request Sense* command additional 8information about the current CHECK CONDITION is returned in the Additional Sense Code field as follows:

Table 5 *Additional Sense codes Supported*

Description	*Code*
No Sense	0x00
Invalid Sense	0x20
Invalid Address	0x21
Invalid Field	0x26
Bad Transfer	0x80
Bad Parameter	0x81

2.7 Read and Read Extended

See descriptions of individual protocols below.

2.8 Write and Write Extended

See descriptions of individual protocols below.

2.9 Seek

Seek will be ignored and will always return *GOOD* status.

2.10 Inquiry

The *Inquiry* command will return the standard inquiry data header along with vendor unique data in the form of text describing the interpreter. This text is *ADOBExxxSCSICHAN(C)1990 ADOBESYS*, where *x* indicates a space. The Peripheral Device Type field will return 0x00 indicating a disk. Removable Media Bit (RMB) will be zero. The Device Type Qualifier is unspecified. The ISO Version, ECMA Version and ANSI-Approved Version fields will all be zero.

2.11 Reserve Unit

Will be ignored and will always return GOOD status.

2.12 Release Unit

Will be ignored and will always return GOOD status.

2.13 Send Diagnostic

Will be ignored and will always return GOOD status.

2.14 Start/Stop Unit

Will be ignored and will always return GOOD status.

2.15 Read Capacity

Read capacity will return all of the address space covering all of the protocols on this LUN. For this protocol, *Read capacity* will return the size of the PostScript interpreter's logical address range in sectors (the largest usable *Logical Block Address* for the protocol at the LUN in question). This will be returned in the *Logical Block Address* area and 512 in the block length area. The PostScript interpreter calculates the largest usable block address according to the following formula:

0x1000 + (size of the SCSI in-band data stream input buffer in bytes)/512.

0x1000 is the address of the in-band data stream, and 512 is the number of bytes per sector. The partial medium indicator (PMI) bit and the *Logical Block Address* fields will be ignored and should be set to zero for future compatibility.

3 Simple Stream Specification

Simple stream provides a simple but functional interface with built-in flow control and out-of-band data capabilities. This stream implements a packet style protocol and can be found on LUN 0 at Logical Block Addresses *0x1000* and *0x1010* for the in-band data and out-of-band data respectively. Any other *Logical Block Address* will either access another protocol or return an error. Use the following 'C' code to define the Logical Block Addresses for the in-band and out-of-band streams.

```
/* Stream byte addresses */
#define SSII_DATA          0x1000*512
#define SSII_OUTOFBAND     0x1010*512
```

Packet headers are used to convey commands between devices. The same structure is used for reading and writing in-band or out-of-band data. Any data should directly follow the packet header and should be padded out with zeros so that the header bytes plus the data bytes plus the padding add up to an even multiple of 512. Upon completion of each command the RIP will return a status byte to the host. The use of each of the fields in the packet header will be described below. The header is arranged as follows:

Table 6 *Packet Header*

Word position	*Content*
Word 0	Data Type
Word 1	Count
Word 2	Buffer Size
Word 3	Flags
Word 4	Sequence
Word 5	Channel
Word 6	Reserved
Word 7	Reserved

Note that each field is a 32 bit word. Care must be taken to insure the correct byte ordering of the header fields when transferring data between the host and RIP. All PostScript interpreters assume the host is "big-endian" or least significant byte first. If the host is "little-endian" then some byte reordering manipulations will need to be done before and after transferring the packet header.

To define the packet header in a “C” program you would use the following structure:

```
typedef struct pkthdr {
 unsigned int DataType;
 unsigned int Count;
 unsigned int bufsize;
 unsigned int Flags;
 unsigned int Sequence;
 unsigned int Channel;
 unsigned int Reserved4;
 unsigned int Reserved5;
} PktHdr, *pkthdr;
```

3.1 Creating a Packet Header

Keeping this structure in mind the following rules should be used when reading and writing a packet header with a host application. A more detailed description of the individual fields will follow.

In-band data stream

- *Normal Data* indicates normal in-band data
- *Normal Data + End of File* implies that *end of file* follows the last data byte in the packet.

Out-of-band data stream

- *Interrupt* will indicate an interrupt is being sent to the RIP.
- *Status* will indicate a status packet is being sent from the RIP.
- *ErrorInfo* will indicate that error information is being sent from the RIP.

Both streams

A detailed description about the decisions to make for filling in the *data type, buffer size, count, flags, sequence, channel and reserved* fields follows.

Data Type

There are five data types supported by this protocol Normal Data, Normal Data+End of File, Interrupt, Status and Error Info. These are the only valid data types found in a packet header currently, other types may be added at a later date. Normal Data and Normal Data + End of File will be bidirectional data types, i.e. both host and RIP will read and write packets of this type. An Interrupt data type will only be written by the host. Status and Error Info data will only be read by the host. All are described as follows:

Table 7 *Data Types*

Type	Code	Action by Host
Normal Data	0x00	Read/Write
Normal Data + End of File	0x01	Read/Write
Interrupt	0x02	Write
Status	0x03	Read
Error Info	0x04	Read

These can be defined in “C” using constants:

```
#define NORMDAT          0x00
#define NORMDATANDEOF    0x01
#define INTERRUPT        0x02
#define PKTSTATUS        0x03
#define ERRORINFO        0x04
```

A data type of Normal Data indicates a read or write on the in-band data stream. Normal Data + End of File indicates that the end of data transfer has arrived and all the data has been sent. A data type of Interrupt indicates that the RIP should stop what its doing. Status, and Error Info indicate out-of-band data containing information about the state of the RIP is being sent to the host.

Buffer Size

The *Buffer Size* field of the packet header of a write request from the host is used by the RIP to determine the amount of buffer space available at the host. Each write request to the RIP will contain the current free space of the host’s input buffer in sectors. The number of sectors specified here represents the total number of bytes divided by 512. This number will be used by the RIP to determine how much data to send to the host on any subsequent reads. In this way The host application writer should take the following points into consideration when choosing the value for the host’s buffer size:

- This value determines an upper bound on the amount of data the PostScript interpreter will send back to the host upon receipt of the next read request. At the time a given read request is received by the PostScript interpreter, there may be less valid data available for output than the host is able to accommodate. In this case the RIP will append to the data a pad of zeroes such that *(header + data + pad / sector size)* equals the number of sectors specified in the SCSI header. If there is more data available than the host can accommodate, the RIP will send back a number of valid data bytes equal to the most recent host buffer size figure i.e.:

  ```
  size = (bytesavailable > lastsent)? lastsent: bytesavailable;
  ```

  Another read request must then be issued to pick up the data bytes which have not yet been read.

- If the host's buffer size is zero, the PostScript interpreter assumes that the host is able to accommodate as much as *(sector size- header or 480)* bytes of valid data, and will send no more than this number of bytes back to the host in a given read request i.e.:

  ```
  size = (bytesavailable == 0) ? (BUFSIZE - HEADERLEN + 1): bytesav-
    ailable
  ```

  A buffer size of less than zero is undefined; negative buffer size codes are reserved for future use.

Until the host issues a write command, the PostScript interpreter has no way of knowing the host's input buffer size. It acts as though the buffer size was zero and the conditions outlined above apply. The PostScript interpreter assumes that the available host buffer size is equal to the *Buffer size* field specified in the header of the last write request issued by the host. If more than one read request is issued between successive write requests, the PostScript interpreter uses this most recent value when determining the amount of valid data to send back for each request as described above.

When reading from the RIP the

is always set by the RIP to be equal to the amount of free space in the input buffer.

Count

The *Count* field will contain the number of data characters, excluding header and padding, to be considered valid in the packet.

Flags

Currently there is a single *flag* implemented. The *Disconnect This Transaction* flag is defined as *0x01*. Unused flag bits should be set to zero, other flags may be defined in the future.

Note that the *Disconnect This Transaction* flag has different implications for host read and host write transactions. In the write case, the write will be disconnected if any processing delay is encountered. In the read case, the transaction will be disconnected if any processing delay is encountered or if the packet would have returned with a count field of zero on a non-disconnected transaction. Note that the RIP decides to disconnect on a transaction by transaction basis and the host should not rely on this behavior in any particular situation. This eliminates the need to poll for PostScript interpreter output on hosts which support *Disconnect/Reselect*. The current PostScript interpreters do not support *Disconnect/Reselect*, although later implementations may.

Sequence

The *Sequence* field is a packet sequence number which should uniquely identify each packet header. When reading the host is guaranteed to receive a unique sequence number from the RIP. In the event of a write retry by the host the RIP will insure that a new sequence number is generated for each transaction. Receiving two packets with the same sequence number at the RIP will generate an CHECK CONDITION status message. When either side generates a sequence number for a header, it generates a number unique within the last 32768 packets. Separate counts are maintained for the in-band and the out-of-band streams.

Channel

The *Channel* field is for future operation with multiple channels. It should be set to zero and will be to zero by the RIP.

Reserved

The two *Reserved* fields will be defined at a later date.

3.2 Packet Header Code Segment

The following code segment builds a packet header in preparation to write to the RIP. The stream type, data type, and data count are parameters,. *pkthdr* is the type described above and buf is a data structure that will hold both the header and the data to be sent.

```
/* Create packet header */
pkthdr = &buf->header; /* Get pkthdr to point at the top of "buf"
  */
    if (stm_descriptor == SSII_DATA) {
    switch(mode) {
    case NORMDAT: pkthdr->DataType = NORMDAT; break;
    case NORMDATANDEOF: pkthdr->DataType = NORMDATANDEOF; break;
    default:
      fprintf(stderr, "Trying to send an out-of-band packet down
  in-band stream\n");
      ResetTTY();
      kill(cpid, SIGTERM);
      exit(0);
      break;
    }
  }
  else if (stm_descriptor == SSII_OUTOFBAND) {
    switch(mode) {
    case INTERRUPT: pkthdr->DataType = INTERRUPT; break;
    default:
      fprintf(stderr, "Trying to send an in-band packet down out-
  of-band stream\n");
      ResetTTY();
      kill(cpid, SIGTERM);
      exit(0);
      break;
    }
  }
  else {
    fprintf(stderr, "Trying to send down an undefined stream\n");
    ResetTTY();
    kill(cpid, SIGTERM);
    exit(0);
  }

  pkthdr->Count = nbytes;
  pkthdr->BufSize = 0;
  pkthdr->sequence += 1;
/* zero the flag,and reserved fields */
  zeroEmptyFlds(pkthdr);
```

4 Using The In-Band Data Stream

The data stream at address 0x1000 handles all in-band data (PostScript language program text). To use this channel without invoking the SCSI flow control it is necessary to find the size of the PostScript interpreter's input buffer. To do this, read either the in-band data stream or the out-of-band stream. The *Buffer Size* field of the returned packet header will contain the current free space in the PostScript interpreter's input buffer. *It is necessary to check the buffer size before each write to determine the buffer space available at the PostScript interpreter.* If any writes issued by the PostScript interpreter are greater than the available buffer size then a CHECK CONDITION status and a BAD PARAM error will be returned

On reading this data stream will return any pending output data from the PostScript interpreter with the header's *Data Type* field being set to *Normal Data* or *Normal Data + End of File*. The *Count* field will contain the count of data characters to be considered valid in the packet. The *Buffer Size* field will contain the current free space in the PostScript interpreter's input buffer. The *Flags* field will contain any flags asserted for the transaction. The *Sequence* field will contain a unique number. See below for more details about reading from the data stream. The out-of-band channel is similar except it will return either *Status* or *Error Info* type data.

See appendix A for code listings that demonstrate how to determine the size of the current buffer as well as writing to the appropriate stream.

Writing to the In-Band Data Stream

Supported Data Types: *Normal Data, Normal Data + End of File*

Text in the form of a PostScript language job should be written to the in-band data stream. Bytes written to this stream will be sent to the PostScript interpreter unaltered. This allows binary data, such as image data, to be written on this stream. To write to this stream, the PostScript interpreter's buffer size should be determined as described in section 3 under *Buffer Size*. A packet header should be formed with data type set to either *Normal Data* or *Normal Data + End of File* as appropriate. The *count* field should be set to the count of data bytes to be sent (not including the header's 32 bytes). The buffer size should be set to the RIP's currently available buffer space. The following code demonstrates in-band writing. It is a procedure that downloads a file to the RIP. Download will first determine if the RIP has any buffer space available. If it does then that buffer size amount is read from the file and downloaded, if it doesn't the host waits until space is available.

```
void download(fd)
int fd; /* File descriptor */
{
  int n, bytesavailable, mode, xfer;
  int temp;
  int waited=0;

  do {
    int count=0;
    char twiddle;

    if ((bytesavailable = RIPinbuf_BytesAvail(NORMDAT)) < MINI-
   MUMXFER)
      waited=1;
/* wait for RIP buffer to become available */
    while (bytesavailable < MINIMUMXFER) {
      switch(count++ & 0x3) {
      case 0:
   twiddle='\\'; break;
      case 1:
```

```
   twiddle='|'; break;
      case 2:
   twiddle='/'; break;
      default:
   twiddle='-'; break;
      }
      fprintf(stderr, "%c", twiddle);
      usleep(BREAKTIME); /* If the interpreter's busy thinking,
   pause to prevent thrashing */
      fprintf(stderr, "\b");
     bytesavailable = RIPinbuf_BytesAvail(NORMDAT);
    }

    if (waited) {
      fprintf(stderr, "!");
      waited=0;
    }

     xfer = (bytesavailable > MAXIMUMXFER) ? MAXIMUMXFER : bytesav-
   ailable;
/* read the data from the file */
     n = read(fd, ThePacket.data, xfer);

     if (n < 0) {
   fprintf(stderr, "error reading from stdin");
   fflush(stderr);
   kill(cpid, SIGTERM);
   exit(0);
     }
/* if we read no data from file send EOF else normdata */
     if (n == 0) mode = NORMDATANDEOF;
     else {
       mode = NORMDAT;
     }
     temp = n;
     while( temp > 0) {fprintf(stderr, "@"); temp -= 4096;}
     fflush(stderr);
/* Now do the write */
     if (mode == NORMDATANDEOF) {
       WriteSCSIStm(scsi, &ThePacket, n, mode);
       fprintf(stderr, "\ndone sending\n");
       isEOF = 0;
       fflush(stderr);
       break;
     }
     else WriteSCSIStm(scsi, &ThePacket, n, mode);


     /* Flush out any error msgs */
     ChkRIPStatus(); } while (n > 0);
```

As demonstrated above the host does not maintain a running total of bytes sent to the RIP. It simply sends the number of bytes available at the RIP until all its data is sent. On intervening reads the RIP assumes that the host buffer size is at least as large as the *buffer size* of the last write. This frees the host application from needing to issue write requests before every read request to

inform the PostScript interpreter that space is still available. The implication is if the host plans to issue multiple reads without intervening writes, it is the host driver's responsibility to ensure that the available host buffer size is at least as large as the last size reported to the PostScript interpreter before issuing another read.

Note that this channel, in this mode, does not support data types other than those mentioned above. An unsupported data type for a given channel will result in the transfer being aborted and a *CHECK CONDITION* status byte being returned on the SCSI bus

Reading from the In-Band Data Stream

Supported Data Types: *Normal Data, Normal Data + End of File*

Output from a PostScript language program being interpreted (not including error and status messages) will appear in the in-band data stream when it is read. The data read from the stream will contain a packet header which has been tailored to the size of the read request on the SCSI bus. The data type will be set to either *Normal Data* or *Normal Data + End of File* as appropriate, the count set to the count of actual data bytes contained in the packet (not including the header's 32 bytes) and the buffer size set to the PostScript language interpreter's input buffer free space. *Sequence* is incremented so as not to repeat a previous number. All other fields are set to the values described in section 3.1

Note that the RIP, in this mode, does not support data types other than those mentioned above. The following code demonstrates generic band reading dependant on the mode passed to the procedure.

```
RIPinbuf_BytesAvail(mode)
int mode; /*indicate the packet's datatype */
{
PPktHdr hdr;
char buf[BUFSIZE+1]; /* the buffer that holds all 512 bytes sent
   back by RIP */
char *databuf; /* he points only to any data sent back */
int rem;
unsigned int stm_descriptor;

  switch(mode)
    {
    case NORMDAT: case NORMDATANDEOF:
      stm_descriptor = SSII_DATA; break;
    default:
      stm_descriptor = SSII_OUTOFBAND; break;
    }
lseek(scsi, stm_descriptor, 0); /* Seek to a non-zero spot */

if (read(scsi, buf, BUFSIZE) != BUFSIZE)
  {
```

```
    fprintf(stderr, " in RIPinbuf_BytesAvail, BUFSIZE=%d\n ",BUF-
   SIZE);
    perror(" read error ");
    ResetTTY();
    kill(cpid, SIGTERM);
    exit(0); /* he's still around somewhere */
  }

hdr = (PktHdr *) &buf[0]; /* Coerce into a packet header */

databuf = (char *) &buf[HEADERLEN];
databuf[BUFSIZE-HEADERLEN+1] = '\0';

if (hdr->DataType == NORMDATANDEOF)
  {
    fprintf(stderr, "\n\rlistener got EOT\n\r");
    isEOF = 1;
    fflush(stderr);
  }
return(hdr->BufSize);
}
```

5 Using The Out-Of-Band Stream

The data stream at address 0x1010 handles all out-of-band data (for example error messages, status messages and interrupts). Currently there is no out-of-band data, although there is an *interrupt* command, defined going from the host to the RIP. If it is used in the future it will be subject to the same constraints and conditions as in-band data described above. See appendix A for code listings that demonstrate how to determine the size of the current buffer as well as writing to the appropriate stream.

5.1 Writing to the Out Of Band Stream

Supported Data Types: *Interrupt*

The write side of the out-of-band stream is used for sending signals to the interpreter. Currently only the interrupt signal is defined for the out-of-band stream. This signal is equivalent to a Control-C on a traditional RS-232 channel. It can be sent by creating a packet header with the data type set to Interrupt, the count set to zero and the buffer size set to the size of the sender’s currently available status/error message buffer space. This packet header should be padded out to 512 bytes with zeros and written to the SCSI bus at the appropriate Logical Block Address. Signals sent in this manner will immediately affect the state of the interpreter even if the in-band data stream buffers are full. Note that this channel in this mode does not support data types other than those mentioned above. An unsupported data type for a given channel will result in the transfer being aborted and a *CHECK CONDITION* status byte being returned on the SCSI bus.

The following procedure is a generic procedure that writes a character to the RIP. It is used in interactive mode to determine the type of transaction with the RIP. The transaction is dependant on the first character of the buffer string.

```
Send(c)
char c;
{
  int i;
  char buf[BUFSIZE];
  int mode;

  switch(c)
    {
    case CTRL_C:
      mode = INTERRUPT;
      WriteSCSIStm(scsi, &ThePacket, 0, mode);
      break;
    case CTRL_D:
      mode = NORMDATANDEOF;
      ThePacket.data[0] = c;
      WriteSCSIStm(scsi, &ThePacket, 0, mode);
      break;
    case CTRL_T:
      /* Send a signal to the child process requesting status */
      kill(cpid, SIGUSR1);
      break;
    default:
      mode = NORMDAT;
      ThePacket.data[0] = c;
      WriteSCSIStm(scsi, &ThePacket, 1, mode);
      break;
    }
}
```

Reading from the Out Of Band Stream

Supported Data Types: *Status, Error Info*

Error messages and status output from a PostScript language program being interpreted can be read from the *out-of-band* stream; what constitutes an error message is dependent on the implementation. Reading from the *out-of-band* stream when there are no pending error messages will return the interpreter's status string (a service traditionally provided by Control-T on an RS-232 channel). The data type will be set to either *Status* or *Error Info* as appropriate. The count will be set to the count of actual data bytes contained in the packet (not including the header's 32 bytes) and the buffer size set to the PostScript interpreter's input buffer free space. The *Error Info* data type has a higher priority than the *Status* data type, therefore, pending error messages at the interpreter will cause the data type to be *Error Info* and the data to contain the error messages. Note that status cannot be specifically requested but is returned as a result of there being no more error messages. All reserved fields are undefined. The data directly follows the packet header and is padded out

with zeros so that the header bytes plus the data bytes plus the padding add up to the requested number of sectors multiplied by 512 bytes. Note that this channel, in this mode, does not support data types other than those mentioned above. The following is an example of a procedure that reads from the out-of-band stream. ChkRIPStatus() is called at fixed intervals by a child process. It performs a read on the out-of-band stream.

```
extern int errno;
char statusbuf[BUFSIZE];
ChkRIPStatus()
{
  PPktHdr hdr;
  char buf[BUFSIZE+1]; /* the buffer that holds all 512 bytes sent
   back by RIP */
  char *databuf;
  int status_len;

  /* Seek to address corresponding to out-of-band stream*/
  lseek(scsi, SSII_OUTOFBAND, 0);

  /* Read the out-of-band stream */
  if (read(scsi, buf, BUFSIZE) != BUFSIZE) {
    if (errno == EINTR) { /* We were interrupted by a status re-
   quest */
      fprintf(stderr, "Out-of-band data stream access interrupted
   by parent process\n\r");
      errno = 0;
      return;
    }
    else {
      perror(" read error in ChkRIPStatus ");
      ResetTTY();
      exit(0);
    }
  }
hdr = (PktHdr *) &buf[0]; /*Coerce buf into a packet header */
  databuf = (char *) &buf[HEADERLEN];
  databuf[BUFSIZE-HEADERLEN+1] = '\0';
  strcpy(statusbuf, databuf); /* Copy databuf to statusbuf */
}
```

Appendix: A - SCSIPS Host Program

Applications for a Unix Host

The following programs are used to build Interface and scsips, two programs that demonstrate the use of the SCSI bus. Segments of these programs were used in the body of the specification to demonstrate various aspects of the protocol. There is disk associated with this specification that contains the following code as well as the makefile to build the applications.

```
/* getargs.c
% Copyright (C) 1990-1993 Adobe Systems Incorporated. All rights reserved
%
% This file may be freely copied and redistributed as long as:
%   1) This entire notice continues to be included in the file,
%   2) If the file has been modified in any way, a notice of such
%      modification is conspicuously indicated.
%
% PostScript, Display PostScript, and Adobe are registered trademarks of
% Adobe Systems Incorporated.
%
% ************************************************************************
% THE INFORMATION BELOW IS FURNISHED AS IS, IS SUBJECT TO CHANGE WITHOUT
% NOTICE, AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY ADOBE SYSTEMS
% INCORPORATED. ADOBE SYSTEMS INCORPORATED ASSUMES NO RESPONSIBILITY OR
% LIABILITY FOR ANY ERRORS OR INACCURACIES, MAKES NO WARRANTY OF ANY
% KIND (EXPRESS, IMPLIED OR STATUTORY) WITH RESPECT TO THIS INFORMATION,
% AND EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES OF MERCHANTABILITY,
% FITNESS FOR PARTICULAR PURPOSES AND NONINFINGEMENT OF THIRD PARTY RIGHTS.
% ************************************************************************
%
*/

#include <stdio.h>
/*
** Stuffs one command line argument per invocation
** in the  array of chars 'line'; returns EOF
** when it's at the end of it's rope
*/

getarg(index, linegth, argc, argv)
int index,/* Index of command line entry to return */
    length,/* Length of 'line' */
    argc;/* Number of command line arguments */
```

```
char *line,/* Array in which to stuff comm line arg */
     *argv[];   /* array of comm line elements */
{
  int i = 0;

  if (index >= argc || length == 0) return EOF;

/*  printf("getarg: length = %d\n", length);*/


  do {
    line[i] = argv[index][i++];
/*    printf("getarg: index = %d, i = %d, line[%d] = %c\n", index, i-1, i-1,
              line[i-1]);*/
  } while ((line[i-1] != '\0') && (i < length));

  return 0;
}

/* interface.c
Scsi stream protocol interface for scsips.c
*/

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include "scsips.h"
#include "packet.h"

extern done;
extern int inSend, inChk;
extern int raw_mode;
extern int scsi;       /* file descriptor pointing at scsi device */
extern int cpid; /* the child process id*/
unsigned int in_band_read, out_band_read; in_band_write,
     out_band_write;
BadLseek() { ResetTTY(); fprintf(stderr, "Bad lseek address!\n"); kill(cpid, SIGTERM); exit (-
   1); }

#define BIGENDIAN  1
/*
 * WriteSCSIStm() sends data down to the scsiinput channel. It makes sure
 * that there is space available in the RIP's input buffers before calling
 * "write".
 */
union longchar {
   unsigned long l;
   unsigned char c[4];
};
unsigned long unflip(val)
unsigned long val;
{
    struct longchar lc;
    lc.c[0] = val >> 24;
    lc.c[1] = val >> 16;
    lc.c[2] = val >> 8;
    lc.c[3] = val;
```

```
    return (lc.l);
}
unsigned long flip(val)
ufprintf(stderr, "Trying to send down an undefined stream\n");
    ResetTTY();
    kill(cpid, SIGTERM);
    exit(0);
  }

  pkthdr->Count = nbytes;
  pkthdr->BufSize = 512;
  pkthdr->Sequence = 0;/*(stm_descriptor == SSII_DATA)
     ? in_band_write++ : out_band_write++;*/
  ZapEmptyFlds(pkthdr);
#if BIGENDIAN
/* fix up fields for the big-ender we're going to. */
   pkthdr->DataType = flip(pkthdr->DataType);
   pkthdr->Count = flip(pkthdr->Count);
   pkthdr->BufSize = flip(pkthdr->BufSize);
   pkthdr->Sequence = flip(pkthdr->Sequence);
#endif

  if (!(stm_descriptor == SSII_DATA || stm_descriptor == SSII_OUTOFBAND)) BadLseek();

  lseek(scsi, stm_descriptor, 0); /* Seek to the right spot */
  totalbytes = nbytes + HEADERLEN;
  ThisBuf = (totalbytes % BUFSIZE == 0) ?
    totalbytes : ((totalbytes/BUFSIZE) + 1) * BUFSIZE;

  if((retval = write(fd, buf, ThisBuf)) != ThisBuf)
    {
      ResetTTY();
      if (inSend) fprintf(stderr, "In send\n");
        else if (inChk) fprintf(stderr, "In Chk_inband\n");
      fprintf(stderr, " 'write' mistakenly  wrote %d bytes, instead of %d bytes\n", retval,
   ThisBuf);
      if (inSend) fprintf(stderr, "In send\n");
        else if (inChk) fprintf(stderr, "In Chk_inband\n");
      perror(" write error"); fflush(stderr);
      kill(cpid, SIGTERM);
      exit(0);
    }else

  return (retval);
}
extern int deafflag;
/*
 * RIPinbuf_BytesAvail() polls the RIP for the number of bytes available
 * in the RIP's input buffer, and returns this value.
 * For performance reasons, it makes more sense to keep grabbing data as
 * long as data is available than to grab only one packets worth at at time.
 */
RIPinbuf_BytesAvail(mode)
int mode; /*indicate the packet's datatype */
{
PPktHdr hdr;
char buf[BUFSIZE+1]; /* the buffer that holds all 512 bytes sent back by RIP */
char *databuf; /* he points only to any data sent back */
```

```
int rem;
unsigned int stm_descriptor;

  switch(mode)
    {
    case NORMDAT: case NORMDATANDEOF:
      stm_descriptor = SSII_DATA; break;
    default:
      stm_descriptor = SSII_OUTOFBAND; break;
    }

if (!(stm_descriptor == SSII_DATA || stm_descriptor == SSII_OUTOFBAND)) BadLseek();

do{
lseek(scsi, stm_descriptor, 0); /* Seek to a non-zero spot */

if (read(scsi, buf, BUFSIZE) != BUFSIZE)
  {
   fprintf(stderr, " in RIPinbuf_BytesAvail, BUFSIZE=%d\n ",BUFSIZE);
    perror(" read error");
    ResetTTY();
    kill(cpid, SIGTERM);
    exit(0); /* he's still around somewhere */
  }

hdr = (PktHdr *) &buf[0]; /* Coerce into a packet header */

databuf = (char *) &buf[HEADERLEN];
databuf[BUFSIZE-HEADERLEN+1] = '\0';

#if BIGENDIAN
   hdr->DataType = unflip(hdr->DataType);
   hdr->BufSize = unflip(hdr->BufSize);
   hdr->Count = unflip(hdr->Count);
   hdr->Sequence = unflip(hdr->Sequence);
#endif
/*  if(hdr->Sequence == in_band_read)
   fprintf(stderr, "seq: %d", hdr->Sequence);*/

if (raw_mode)
  {
    DsplyWithCR(databuf);
  }
else /* called from onalarmcooked() */
  {
    fwrite(databuf, 1, hdr->Count, stdout);
    fflush(stdout);
  }

if (hdr->DataType == NORMDATANDEOF)
  {
   fprintf(stderr, "\n\rlistener got EOT\n\r");
    done = 1;
    fflush(stderr);
  }
}while ((hdr->Count == (BUFSIZE-HEADERLEN)) && (mode == NORMDAT));
return(hdr->BufSize);
}
```

```
/*
 * ChkRIPStatus() is called at fixed intervals by a child process.
 * It performs a read on the out-of-band stream, and checks the type
 * of the returned packet. If the returned packet is a status packet,
 * a global status string is updated. If the returned packet is
 * an error packet, the error msg is displayed immediately, and the
 * status message is updated as well.
 */
extern int errno;
char statusbuf[BUFSIZE];
ChkRIPStatus()
{
  PPktHdr hdr;
  int readval;
  char buf[BUFSIZE+1]; /* the buffer that holds all 512 bytes
         sent back by RIP */
  char *databuf;
  extern int deafflag;
  /* Seek to address corresponding to out-of-band stream*/
  if(lseek(scsi, SSII_OUTOFBAND, 0)<0) {
      if (inSend) fprintf(stderr, "In send\n"); else if (inChk) fprintf(stderr, "In Chk_in-
   band\n");
    perror(" Bad lseek ");
    ResetTTY();
    exit(1);
    }

  /* Read the out-of-band stream */
  if ((readval=read(scsi, buf, BUFSIZE)) != BUFSIZE) {
    if (errno == EINTR) { /* We were interrupted by a status request */
      fprintf(stderr, "Out-of-band data stream access interrupted by parent process\n\r");
      errno = 0;
      return;
    }
    else {
      fprintf(stderr, "readval=%d\n", readval);
      perror(" THIS read error ");
      if (inSend) fprintf(stderr, "In send\n"); else if (inChk) fprintf(stderr, "In Chk_in-
   band\n");
      ResetTTY();
      exit(0);
    }
  }
  hdr = (PktHdr *) &buf[0]; /* Coerce this guy into a packet header */
#if BIGENDIAN
        hdr->DataType = unflip(hdr->DataType);
        hdr->BufSize = unflip(hdr->BufSize);
        hdr->Count = unflip(hdr->Count);
        hdr->Sequence = unflip( hdr->Sequence);
#endif

  databuf = (char *) &buf[HEADERLEN];
  databuf[BUFSIZE-HEADERLEN+1] = '\0';
  strcpy(statusbuf, databuf); /* Copy databuf to statusbuf */
  if(hdr->Sequence == out_band_read)
   fprintf(stderr, "Sequence now supported-outband\n");
  if (hdr->DataType == ERRORINFO) {
```

```
    if (raw_mode) DsplyWithCR(statusbuf);
    else {
      fprintf(stderr, "%s", statusbuf);
      fflush(stderr);
    }
  }
}

/* scsips.c
Utility for sending jobs to, or entering PostScript interactive mode with RIP.
Currently set up so that X will be printed to stdout b/w packets.  This
makes the task of verifying that the PS flush operator is forcing the flush
behavior required to make Mac applications feeding PS to the rip over scsi
work. See the SHOW_FLUSH #define in interface.c.
*/

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <setjmp.h>
#include "scsips.h"
#include "packet.h"

#define LINESIZE 64
#define BIGENDIAN 0
#if BIGENDIAN
#define SCSIPORT "/dev/rrz1c"
#define BREAKTIME .1
#define usleep sleep
#else
#define SCSIPORT "/dev/rsd2c"
#define BREAKTIME 100000
#endif
char *progname;
int done, almostdone; /* Flag set by GetRIPinbuf forblemumble*/
int scsi; /* fid for the scsi port */
int cpid; /* child processes id numbers*/
int spawned_child; /* flag indicating whether you've already spun off the
           process for monitoring output from the RIP */

int doingfiles;
extern char statusbuf[BUFSIZE];
char *usage = "[-i -b -s -g -t] [file ...]";

static Pkt ThePacket;
extern unsigned int in_band_read, in_band_write,
     out_band_read, out_band_write;
/* Debugging flags */
int debug;
int glenflag, deafflag, separate_data, ttyflag, exper;
int inChk;
int inSend;
int badsend;

main(argc, argv)
int argc;
char *argv[];
{
```

```
  int f1;
  char line[LINESIZE];
  int i, gointeractive, x, y;
  FILE *fopen();
  char *thePort;
  void download();
  extern onintr();

  progname = argv[0];

  i= done= gointeractive=0;

  /* Assert debug mode */
  debug = 0;
  in_band_read = in_band_write = 1;
  out_band_read = out_band_write = 1;

  /* open the scsi port in rw mode */
  if ((scsi = open(SCSIPORT, 2)) == -1)
    {
      fprintf(stderr, "can't open %s for read/write", thePort);
      exit(1);
    }

  while (getarg(++i, line, LINESIZE, argc, argv) != EOF)
    {
      if (!doingfiles) /* Still processing command line flags. */
   {
     if (line[0] == '-') /* command line switch */
       {
         switch(line[1])
     {
        case 'g':/* always send 15328 bytes */
          glenflag = 1; break;
        case 't':
          ttyflag = 1;/* getarg(++i, line, LINESIZE, argc, argv);*/
#if 0
        strcpy(mybuf, line[2]);
   printf("%s %s\n", line[0],mybuf);
        thePort = mybuf;
#endif
        break;
        case 's':/* separate data from eof pkt */
          separate_data = 1; break;
        case 'd':/* debug channel */
          debug = 1; break;
        case 'h':/* don't listen to back channel */
          deafflag = 1; break;
   case 'i': /* interactive mode */
     gointeractive = 1; break;
   case 'X':/* Send big bunch of EOFs, then a string */
     exper = 1;
     break;
   case 'b':/* bad method of sending--sends half */
     badsend = 1;
     break;
   default:
     fprintf(stderr, "Usage: %s\n", usage);
```

```
    exit(1);
    break;
  }
      }
    else /* filename */
      {
        doingfiles = 1;
      }
  }
     if (doingfiles) /* Download files */
  {
    if (line[0] == '-') /* Not accepting flags; only filenames */
      {
        fprintf(stderr, "Usage: %s\n", usage);
        exit(1);
      }
    /* open up the file to be sent down */
    if ((f1 = open(line, 0)) == -1)
      {
        fprintf(stderr, "\n\rcan't open %s\n\r", line);
        exit(1);
      }
    fprintf(stderr, "\n\rsending %s\n\r", line);
    download(f1);
  }
  if(exper){
     /* send a bunch of EOF packets */

     /* then send a string followed by 1 eof pkt */
  }
    }
  /* Sending stdin? */
  if (!gointeractive && !doingfiles)
    {
      fprintf(stderr, "\n\rsending stdin\n\r");
      download(fileno(stdin));
    }

  if(exper){
#if 0
int bytesavailable;
        bytesavailable = RIPinbuf_BytesAvail(NORMDAT);
        fprintf(stderr, " #%d# ", bytesavailable);
        bytesavailable = RIPinbuf_BytesAvail(PKTSTATUS);
        fprintf(stderr, " #%d# ", bytesavailable);
#endif
     doexper();
  }
  /* Now that any files to be sent are there, see if you need to go
     interactive */
#if 1
    x = RIPinbuf_BytesAvail(PKTSTATUS);
  y = RIPinbuf_BytesAvail(NORMDAT);
  if(x != y) printf("x = %d, y = %d\n", x, y);
#endif
  if (gointeractive)
    {
      /* FIRST: Put the tty into raw mode */
```

```
      if(!ttyflag)
     InitTTY();
      /*re-map signals to restore tty settings if process is zapped */
      signal(SIGUSR1, SIG_IGN); /* Just ignore this signal if it's sent
             to yourself */
      if (signal(SIGINT, SIG_IGN) != SIG_IGN)
   signal(SIGINT, onintr);
      if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
   signal(SIGQUIT, onintr);
      if (signal(SIGKILL, SIG_IGN) != SIG_IGN)
   signal(SIGKILL, onintr);

/*    if(!deafflag){*/
      if ((cpid=fork()) == 0)
   Get_RIP_data(); /* The Kid */

      /* FINALLY: do it! */
      interact();
    }
  else /* Finish up */
    {
/* keep reading data fed by the RIP, until we receive the EOF */
      while(!done) {
     usleep(BREAKTIME);
     RIPinbuf_BytesAvail(NORMDAT);
   }
     Chk_inbandstm();
      fflush(stderr);
      exit(0);
    }
}

doexper()
{
   register int mode, i;
#if 0
   for(i = 0;i < 400;i++){
   Chk_inbandstm();
      WriteSCSIStm(scsi, &ThePacket, 0, NORMDATANDEOF);
   }
#endif
   i = 0x65;
   mode = NORMDAT;
      ThePacket.data[0] = i;
      WriteSCSIStm(scsi, &ThePacket, 1, mode);
}

interact() /* Driver for interactive mode over scsi */
{
  int c;
  int exitstate, carryover;
  int i;

  fprintf(stderr, "Going interactive... !<ESC> to exit.\n\r");

  exitstate = carryover = 0;
  while(1)
    {
```

```
      c=getchar();

      if (exitstate)
   {
     if (c == ESC )
       {
         fprintf(stderr, "%c%c", '\n','\r');
         if(!ttyflag)
        ResetTTY();
         kill(cpid, SIGTERM);
         exit(0);
       }
   }

      switch(c)
   {
   case '!': /* To leave interactive mode, type "!<escape>" */
     if (exitstate == 1)
       {
         exitstate = 0;
         carryover = 1; /* got 2 "!"s in a row; signal that...*/
       }
     else
       { /* Just saw a "!" */
         exitstate = 1;
       }
     break;
   default: /* Regular character to be sent to printer */
     if (carryover == 1)
       {
         carryover = 0; /* reset flag */
         Send('!');
       }
     Send(c);
/*   fprintf(stderr, "|"); fflush(stderr);*/
     break;
   }
    }
}

int raw_mode;

Get_RIP_data()
{
  int onalarm();
  extern int dsply_status();
  raw_mode = 1;
  signal(SIGUSR1, dsply_status);
  while (1) {
    usleep(BREAKTIME);
    Chk_inbandstm();
  }

}

dsply_status()
{
printf("yup");
```

```
  signal(SIGUSR1, dsply_status);
  if (raw_mode)
    {
      DsplyWithCR(statusbuf);
    }
  else
    {
      fprintf(stderr, "%s", statusbuf);
      fflush(stderr);
    }
}

Chk_inbandstm()
{
  inChk=1;
/*  if (inSend) return;*/
  if (debug) fprintf(stderr, "*");
  ChkRIPStatus();
  RIPinbuf_BytesAvail(NORMDAT);
  inChk=0;
}

DsplyWithCR(b)
char *b;
{
#if 0
  int strlen();
#endif
  int len=strlen(b);
  int count;

  for (count=0; count<len; count++)
    {
      fprintf(stderr, "%c",b[count]);
      if (b[count] == '\n')
   fprintf(stderr, "%c", '\r');
    }
  fflush(stderr);
}

Send(c)
char c;
{
  int i;
  char buf[BUFSIZE];
  int mode;


  inSend=1;

  switch(c)
    {
    case CTRL_C:
      mode = INTERRUPT;
#if 0
      ThePacket.data[0] = c;
#endif
      WriteSCSIStm(scsi, &ThePacket, 0, mode);
```

```
      break;
    case CTRL_D:
      mode = NORMDATANDEOF;
      WriteSCSIStm(scsi, &ThePacket, 0, mode);
      break;
    case CTRL_T:
      /* Send a signal to the child process requesting status */
      kill(cpid, SIGUSR1);
      break;
    default:
      mode = NORMDAT;
      ThePacket.data[0] = c;
      WriteSCSIStm(scsi, &ThePacket, 1, mode);
      break;
    }

  inSend=0;
}

onintr()  /* Clean up if interrupted */
{
  ResetTTY();
}

void download(fd)
int fd; /* File descriptor */
{
  int n, bytesa, bytesavailable, mode, xfer;
  int temp, flipper;
  int waited=0;
  char stage[MAXDATA];

  do {
    int count=0;
    char twiddle;

   if(badsend){
     flipper = (flipper > 1) ? 0 : 1;
   if(flipper){
    if ((bytesavailable = RIPinbuf_BytesAvail(PKTSTATUS)) < MINIMUMXFER){
      bytesa= RIPinbuf_BytesAvail(NORMDAT);
/*if(bytesavailable != bytesa) printf("&"); */
      waited=1;}
     }
   }else {
    if ((bytesavailable = RIPinbuf_BytesAvail(NORMDAT)) < MINIMUMXFER)
      waited=1;
   }

    if (glenflag) {
      while (bytesavailable != 30729) {
        /* usleep(100000); */
       ChkRIPStatus();
        bytesavailable = RIPinbuf_BytesAvail(NORMDAT);
}
      bytesavailable = 15328;
    }
    else {
```

```
    while (bytesavailable < MINIMUMXFER) {
      switch(count++ & 0x3) {
      case 0:
  twiddle='\\'; break;
      case 1:
  twiddle='|'; break;
      case 2:
  twiddle='/'; break;
      default:
  twiddle='-'; break;
      }
      fprintf(stdout, "%c", twiddle);
      usleep(BREAKTIME);
      fprintf(stdout, "\b");
      ChkRIPStatus();
      bytesavailable = RIPinbuf_BytesAvail(NORMDAT);
    }
    } /* not glenflag case */

    if (waited) {
      fprintf(stderr, "!");
      waited=0;
    }

    xfer = (bytesavailable > MAXIMUMXFER) ? MAXIMUMXFER : bytesavailable;
  if(badsend)
     xfer >>= 1;

    n = read(fd, ThePacket.data, xfer);

    if (n < 0)
      {
      if(!ttyflag)
       ResetTTY();
      perror(" error reading from stdin"); fflush(stdout);
      kill(cpid, SIGTERM);
      exit(0);
      }
if(!separate_data){
    if (n == 0) mode = NORMDATANDEOF;
      else { mode = NORMDAT; }
  } else {
       if (n == 0) almostdone = 1;
      mode = NORMDAT;
  }/* separate_data */

    temp = n;
#if 0
    while(temp > 0) {fprintf(stderr, "@"); temp -= 4096;}
#endif
    fflush(stdout);

    if (mode == NORMDATANDEOF) {
      WriteSCSIStm(scsi, &ThePacket, n, mode);
      fprintf(stderr, "\ndone sending\n");
      fflush(stderr);
      break;
    }
```

```
    else {
WriteSCSIStm(scsi, &ThePacket, n, mode);}

    /* Flush out any error msgs */
    ChkRIPStatus();} while (n > 0);
   if(separate_data){
     WriteSCSIStm(scsi, &ThePacket, 0, NORMDATANDEOF);
       fprintf(stderr, "\ndone sending\n");
       fflush(stderr);
   }
  ChkRIPStatus();
}


/* tty.c
*/

#include <stdio.h>
#include <sys/ioctl.h>

struct sgttyb old_tty;
static int inittty;

InitTTY() /* Puts tty in raw mode */
{
   struct sgttyb tty;

   if (gtty(fileno(stdin), &old_tty) == -1)
   {
      printf("gtty failure\n");
      exit(1);
   }

   tty = old_tty;
   tty.sg_flags |= RAW;
   tty.sg_flags &= ~ECHO;
   if (stty(fileno(stdin), &tty) == -1)
   {
      printf("stty failure\n");
      exit(1);
   }
   inittty = 1;
}

ResetTTY()
{
   if(inittty)
   stty(fileno(stdin), &old_tty);
}
```

Index

S

T

U

W