Terminal Communication under UNIX: The e835 High Voltage Control Program

Robert McTaggart
The Pennsylvania State University

Jonathan Streets
Fermilab

December 8, 1995

1 Introduction

One of the problems confronting the upcoming e835 experiment, is how one communicates to a LeCroy 1440 High Voltage System from a Silicon Graphics Indy computer. Basically this means replacing a VAX Fortran program with routines written in C to do the opening of, writing to, and reading from a terminal line.

The e760 program was written in VAX Fortran with SYS$QIO routines to communicate over the RS232 port. Once the user chose a command, the program first converted it into a form the 1440 could understand. Most of the code is dedicated to this assignment, and this did not have to be changed for the SGI.

After this alteration has taken place, the program issued a SYS$QIOW request to write the command to the 1440, one to read the command echo, and one to read each response from the 1440. One nice feature of SYS$QIOW is that it permits one to read with a timeout, so the program will not hang if the 1440 has nothing further to say. The standard terminal driver on the SGI cannot perform a read with a timeout: This must be done by the programmer.

2 Test Setup in Wilson Hall

A LeCroy 1440 High Voltage System was located in the e835 work area on the 10th floor of Wilson Hall. Since the 1440 runs on 220 Volts, a transformer was manufactured so that the 1440 could function in Wilson Hall. Connecting the 1440 to the SGI console was a 16 conductor cable and a RS232-C 25-pin connector. This cable was put in serial port 3 on fnpx04, /dev/ttyd3. The RS232-C format the 1440 uses is[3]

8 Bits
1 Stop Bit
No Parity.

The LeCroy manual shows how to build the connection between the cable from the 1440 using RS232-C format to the cable going to an SGI console using RS232-D format[3].

To see whether the 1440 was functioning normally, we connected it to a dummy terminal and manually typed in the commands which could be understood by the 1440(in capital letters). Thus we also got a good idea of what responses to expect from the 1440 when the program was run. For example, when we typed 'M 1' the 1440's response was 'Mainframe 1 responding'.

To test whether the serial port was functioning properly, we talked to the 1440 directly through the device by turning our window into a dummy terminal via the 'cu' command[1], which calls another terminal and used /dev/ttyd3 as

[1]See Appendix D.

the communication line. In both instances, the 1440 returned the command echo and the appropriate response to a command.

3 Installation at AP-50

These tests were repeated on the group of 1440 mainframes located in the AP-50 counting house using fn835x, from which the run control of the experimental data acquisition for e835 will take place.

Each mainframe was tested individually with an ADM terminal, and the 1445 controllers were replaced if the mainframe did not respond at all(i.e. there was no response on the screen when commands were typed at the keyboard). The replacement controllers had to be reset from the default setting of 9600 baud, whereas the experiment will be using 4800 baud. Any bad cables connecting consecutive mainframes were identified and discarded.

The mainframes are daisy-chained in the sequence 1, 6, 4, 2, 3, 5. The cable reaching under the floor from fn835x to the mainframes is connected to mainframe 1. The serial port on the SGI requires a special attachment to convert the 25 pin connector to the number of pins on the serial port. The same attachment used during tests in Wilson Hall is being used in AP-50.

Tests were performed using both available serial ports on fn835x, namely /dev/ttyd1 and /dev/ttyd2 (labelled '1' and '2' on the back of the machine). For both serial ports, 'cu' was applied and fn835x acted just like the dummy terminal had before.

However, responses of the 1440 to commands sent from the program often did not come back to the program. This happened even though the serial port was set up in exactly the same way as when 'cu' successfully functioned. When 'cu' was used immediately after the program was exited, responses to the commands sent from the program appeared after a return.

The problem was solved when calls to flush the buffer were replaced by additional reads of the 1440(until nothing remained in the buffer). Specifically, this is required after voltages are read from a series of channels or when a voltage is written to a channel. It appears that a UNIX flush does not do the same thing as a VAX flush did previously in e760. The serial port's attributes were set up in the same way as they were in the tests in Wilson Hall. So 'cu' functioned with its own set of terminal characterisitics.

After problems with the serial port were addressed, each command used in the program was tested and any necessary corrections were made. The most serious error in this phase was that when the fortran code opened a file that did not exist previously, the error of 'file does not exist' was carried over to the C-routines that do the actual reading and writing. The C-routines stop the program when 'errno', or the error code, is non-zero. This was rectified by setting 'errno' to zero before a read or a write.

In editing the fortran code, it was also revealed that SGI FORTRAN does not recognize the carriage-control characters in the beginning of FORMAT statements by default. To use this option, one must type in the beginning of the program:

open(unit=6,carriagecontrol='FORTRAN').

Also, the name of the printer used to print a hard copy is set within the subroutine PRINT_READ.

The following sections detail the conversion of the VAX version of the high voltage control program to its current UNIX state.

4 Opening a Terminal Line

4.1 Moving from Fortran to C

When VAXes ruled the world, the following set of commands were used to open a terminal line to talk to the 1440:

```
ISTAT=SYS$ASSIGN(TTNAM,ICHAN,,)
IF(ISTAT.NE.1) THEN
  TYPE *,' can't assign channel to serial line'
  CALL LIB$STOP(%VAL(ISTAT))
ENDIF
```

Here TTNAM is the serial line device name, and ICHAN is the number of the channel that is assigned. Because there is no such thing as SYS$ASSIGN in UNIX, we must replace it and all other such system services with analogous routines written in C.

We have placed all of these C routines in one file so that all of them can share the same include files. If your program is not compiling properly, check the manual pages available on the SGI to see which include files are needed for a particular C-function. For example in our file we include

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <ctype.h>
#include <stropts.h>
#include <string.h>
.
```

In the fortran code we replace the SYS$ASSIGN function with a call to a subroutine written in C.

```
CALL IO_OPEN(FD)
```

'FD' is a file descriptor through which the terminal line is referenced in any open, write, or read, and this is passed back and forth from Fortran to C. Calling a subroutine written in C from within Fortran requires a couple of adjustments. First, the name of the routine in C is one character longer, i.e. an underscore ('_') must be added. Next, all variables being passed back to fortran must begin with an asterisk, '*'. Each variable from Fortran is thus made into a 'pointer'. '*fd' points to the location of the variable 'fd'. The program will not compile unless both of these things are done.

4.2 Changing Terminal Attributes of the SGI

Appendix A contains the C routine that opens a terminal line. Opening a terminal line requires two different procedures. First, the file descriptor must be assigned to the terminal line. Then the terminal attributes of the SGI device, i.e. /dev/ttyd3, must be changed to communicate to the 1440.

An open statement assigns the file descriptor to the terminal line. Since we are opening /dev/ttyd3, fd is defined as 3(both 1 and 2 are reserved, and 3 is assigned to the first available serial port). The flag O_RDWR means that we want to open the terminal line such that we can read from or write to it.

termios_p points to a structure called termios. The termios structure basically contains all the information about the attributes for the device which is referenced by the file descriptor, fd. The general procedure for changing the attributes of the device is (a) read the attributes of the device into the termios structure, (b) prepare which attributes are to be changed, and (c) change the termios structure. (a) is usually done by a tcgetattr command, and (c) is accomplished by tcsetattr. Each of these two commands has two arguments: the file descriptor(*fd), and the pointer to the termios structure (&termios_p). The latter must be preceded by an ampersand, '&', for the program to compile.

The termios structure consists of four fields. Each attribute is represented by a flag that influences one of these four fields. To add an attribute to the structure, a flag is bit-or'ed with its respective field. To remove an attribute, the flag's inverse is bit-and'ed with the respective field. The c_iflag field describes basic terminal input control, the c_oflag field specifies the system treatment of output, c_cflag describes the hardware control of the terminal(i.e. characters like EOF), and c_lflag field is used by the line discipline to control terminal functions[2]. The correspondence between the name of the flag and the actual numerical value is determined in one of the include files.

Before changing different terminal characteristics of the SGI, it may be helpful to find out what the default settings for the device in question are. This may be accomplished by typing

stty -a < /dev/ttyd3.

For the SGI used in Wilson Hall, these default characteristics are[2]:

speed 9600 baud; line = 1; intr = DEL; quit =^\; erase = ^H; kill = ^U; eof = ^D; eol = ^@ ; old-swtch = ^@ ; susp = ^@ lnext = ^V; werase = ^W; rprnt = ^R; flush = ^O; stop = ^S; start = ^Q; dsusp =^@ -parenb -parodd cs8 -cstopb hupcl cread clocal -loblk -tostop -ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc ixon ixany -ixoff -imaxbel isig icanon -xcase echo echoe echok -echoke echoctl -echonl -noflsh -flusho -pen din opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3.

Something similar is obtained when the default setting of /dev/ttyd2 on fn835x is examined. Newer SGI machines have slightly more characteristics available for use.

In order for the SGI to communicate properly with the 1440, the echo characteristic of the SGI must be turned off(the rest of the echo* flags may remain in their default settings). Otherwise, there is an infinite echo loop between the SGI and the 1440 mainframe, i.e. the command you write to the 1440 is continuously repeated when the command is echoed back[1]. So we turn off the flag ECHO residing in the c_lflag field.

The other important characteristic to change is the baud rate, which resides

[2] A '-' indicates that the flag is off.

in the c_cflag field. In the default case, the baud rate for the device is set to 9600. For this test we chose to manually set the baud rate of the 1440 to 4800. Instead of or'ing or and'ing bits, we used cfgetispeed and cfgetospeed to obtain the input and output baud rates of the device, and set the new baud rate with the cfsetispeed and cfsetospeed commands.

When 'cu' is used to connect /dev/ttyd2 of fn835x with the 1440's in AP-50, the following characteristics are found:

speed 4800 baud; line = 1; intr = DEL; quit = ^\; erase = ^H; kill = ^U; eof = ^A; eol = ^@; old-swtch = ^@ ; susp = ^@ lnext = ^V; werase = ^W; rprnt = ^R; flush = ^O; stop = ^S; start = ^Q; dsusp = ^@ -parenb -parodd cs8 -cstopb hupcl cread clocal -cnew_rtscts -loblk -tostop ignbrk -brkint ignpar -parmrk -inpck istrip -inlcr -igncr -icrnl -iuclc ixon -ixany ixoff -imaxbel -isig -icanon -xcase -echo -echoe -echok -echoke -echoctl -echoprt -echonl -nofl sh -flusho -pendin -opost -olcuc -onlcr -ocrnl -onocr -onlret -ofill -ofdel

The terminal attributes of the program running on fn835x are:

speed 4800 baud; line = 1; intr = DEL; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = ^@; old-swtch = ^@ ; susp = ^@ lnext = ^V; werase = ^W; rprnt = ^R; flush = ^O; stop = ^S; start = ^Q; dsusp = ^@ -parenb -parodd cs8 -cstopb hupcl cread clocal -cnew_rtscts -loblk -tostop -ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc ixon ixany -ixoff -imaxbel isig icanon -xcase -echo echoe echok echoke echoctl -echoprt -echonl -noflsh -fl usho -pendin opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3

In each of the last two cases, the 'echo' flag has been turned off, and both sets work equally well.

5 Writing to the 1440

The original high voltage control program wrote commands in the following manner to the 1440:

```
    CALL SYS$QIOW(,%VAL(ICHAN),%VAL(IO_WRITE),IOSB,,,
C %REF(SEND),%VAL(LEN),,%VAL(IWFCC),,)

   IF(IOSB(1).NE.1) THEN
      TYPE *,' ************ error writing to 1440'
      CALL SYS$DASSGN(%VAL(ICHAN))
      CALL LIB$STOP(%VAL(IOSB(1)))
   ENDIF.
```

SYS$QIOW is used to I/O requests to a particular device with a timeout. Here ICHAN is the channel being addressed, SEND is the command written to that channel, and LEN is the length of SEND. In the Fortran, this SYS$QIOW is replaced by this routine shown in Appendix B,

```
CALL IO_WRITE(FD,LEN,SEND).
```

The first part of this routine extends the length of the string by one, places a newline character('\n') at the end of the string, and then sets all the remaining characters in the buffer to zero. This makes sure that (a) the 1440 gets a carriage return, and (b) that parts of strings sent in a previous write are not included in this one. Hence the 1440 receives a clear and concise command.

The write command itself returns the number of bytes written if successful, or -1 if unsuccessful. Note that if fd, send, and len were not being passed back to Fortran, the write would have the arguments fd, *send, and len. We then subtract one from the length, since we didn't want the newline character to be counted. This last step is somewhat arbitrary, since we are only counting bytes to make sure that the correct response has been received.

6 Reading from the 1440

Originally the Fortran code read the 1440 in the following manner:

```
      GET=' '
      LTERM(2)=%LOC(LFTERM) !<LF> is input terminator
      CALL SYS$QIOW(,%VAL(ICHAN),%VAL(IO_READ.OR.IO_TIME),IOSB,,,
    C %REF(GET),%VAL(MLEN),%VAL(NSECS),LTERM,,)

      IF (IOERR.NE.0) THEN
        PRINT *,'ERROR IN READING THE 1440'
        CALL FLUSH
        RETURN 1
      ENDIF

      IF(IOSB(1).NE.1) THEN
        IF(IFLAG.EQ.1.AND.IOSB(1).EQ.'22C'X) GO TO 10
        CALL LIB$SIGNAL(%VAL(IOSB(1).AND.'FFFB'X))
        CALL FLUSH
        RETURN 1
      ENDIF
10 LENG=IOSB(2)
'
```

where ICHAN is the channel being read, GET is the string received from the 1440, MLEN is the maximum length of GET, LENG is the actual length of GET, and NSECS is the amount of time that the program will wait for a response.

All of this is replaced by

```
CALL IO_READ(FD,GET,IOERR,LENG,MLEN,NSECS),
```

which is shown in Appendix C.

When a read command is placed inside a while statement, the SGI tries to read as many sequential lines from the 1440 as possible. So if the 1440 has nothing else to say, this read will hang, i.e. the read will keep waiting for something else to come from the 1440 and will not allow the program to continue.

Another problem is dealing with the newline characters that the 1440 transmits. When the 1440 sends back the command echo, that string ends with a newline character. Then the 1440 proceeds to transmit 0-2 more lines that contain a single newline character. Next the 1440 sends back its response to the command just written, which also ends with a '\n'. No further newline characters after a response means the 1440 is done responding. This behavior seems to depend on the command that was sent. For example, no extra newline characters appear after a 'mainframe' command, but at least one such character occurs after a voltage has been written.

The Fortran is structured so that one read is done for each line of response. Whereas SYS$QIOW allowed you to read the response, tell you that the lone '\n' ended that response, and performed a read with a timeout to return to the program when the 1440 was finsihed processing your command, UNIX has no read with a timeout. So one must be very explicit in how newline characters and an empty buffer for the 1440 are addressed.

The routine io_read_ manages to do all three of these tasks performed by SYS$QIOW. The beginning of the while statement first checks to see whether the last line read in the while loop is the final response from the 1440. A string that is only a lone '\n' signifies the end of this read, and we return to the Fortran to request an additional read.

The ioctl[3] command , ioctl(*fd,FIONREAD,&nsub), finds out how many bytes are waiting to be read from the 1440 and places the answer into the integer nsub. If nsub is zero, we wait for a time designated by nsecs to allow the 1440 time to respond. If there is still nothing there, we break the read and go back to the Fortran. It may be the case that this interval of nsecs may have to be increased for some commands and for some individual 1440's, since the response time of the 1440 may depend on the command or on which 1440 you are using. In any case a few seconds is an acceptable wait. Hence this ioctl command tells us if the 1440 is empty. If there are greater than zero bytes waiting, we read this string into the array 'temp', and then copy 'temp' into the array 'get'.

[3]Other ioctl calls perform many other functions, including setting the terminal attributes.

So instead of reading with a timeout, we check to see if the buffer is empty, wait for a while if it is, and then read out the buffer if and only if it is non-empty. If a string consists only of a '\n', we go back to the Fortran. We keep reading the 1440 until nothing or only a '\n' is there.

7 Closing the Terminal Line

To close the serial line we replace the Fortran statement

```
CALL SYS$DASSGN(%VAL(ICHAN))
```

with the C routine

```
void io_close_(int *fd)
{
        close (*fd);
        return;
}.
```

8 Conclusions

We have successfully replaced the VAX system services once used by the Fortran code by routines written in C that do the same functions of opening, reading, and writing to a terminal line and allow the program to work on a UNIX platform. One must set the terminal characteristics of the SGI to match the requirements of the 1440, turn off the SGI echo, and figure out when a read of the 1440 is finished. Since the SGI is much faster than the 1440, the SGI must be forced to wait until the 1440 is ready to give a response. The length of this wait is not set in stone, and appears to depend on the command that was sent.

The program is now operational in AP-50. In the future, adjustments in the program will have to be made so that voltages will be stored and restored from the e835 database instead of from a file in the ../bin directory. If certain code is added to the program, it will run from a button-menu in the e835 run control.

In closing we would like to express our gratitude for the assistance provided by Hans Jostlein, Candies Kastner, and Jose Marques.

References

[1] Private Conversation, K.B. Lee (E831), July 5, 1995.

[2] *termio* manual page on fnpx04.fnal.gov.

[3] LeCroy 1440 High Voltage System Operator's Manual, 1985.

A How to Open a Terminal Line

```
int io_open_(int *fd)
{
        struct termios termios_p;
        speed_t speed;
        int status;
#define REQ_SPEED B4800
        errno=0;
        *fd=open(”/dev/ttyd3”,O_RDWR);
        if(errno != 0)
        perror(“io_open: open(): “);
status= tcgetattr(*fd,&termios_p) ;
termios_p.c_lflag= termios_p.c_lflag & ˜ECHO | IEXTEN ;
status= tcsetattr(*fd,TCSANOW,&termios_p);
if(tcgetattr(*fd,&termios_p)!=0)
        perror(”io_open: tcgetattr()”);
speed = cfgetispeed(&termios_p);
cfsetispeed(&termios_p,REQ_SPEED);
speed = cfgetospeed(&termios_p);
cfsetospeed(&termios_p,REQ_SPEED);
if(tcsetattr(*fd, TCSAFLUSH , &termios_p)!=0)
        perror(”io_open: tcsetattr()”);
        speed = cfgetispeed(&termios_p);
        speed = cfgetospeed(&termios_p);
        return(*fd);
}
```

B How to Write to the 1440

```
void io_write_(int *fd,int *len,char *send)
{
        int nbytes;
        errno=0;
        send[*len] = (char) 0x0a;
        *len=*len+1;
        memset(&send[*len],0,40-*len);
        nbytes=write(*fd,send,*len);
        if(errno != 0)
          perror("io_write: write(): ");
        *len=*len-1;
        return;
}
```

C How to Read from the 1440

```
void io_read_(int *fd,char *get,int *ioerr,int *nbytes,int *ntoread,int *nsecs)
{
        struct termios termios_p;
        char temp[40];
        int nsub;
        int wait_longtime;
        memset(get,0,40);
        *nbytes=0;
        errno=0;
        temp[0] = '\0';
        wait_longtime=1;
        while(temp[0]!='\n'){
            sginap(10);
            ioctl(*fd,FIONREAD,&nsub);
            if(nsub == 0){
              if(wait_longtime==1){
                sleep(*nsecs);
                wait_longtime=0;
              }
              else{
                break;
              }
            }
            else{
              memset(temp,0,40);
              nsub = read(*fd,temp,*ntoread);
              if(errno!=0)
                perror("io_read: read():");
              strcat(get,temp);
              get[strlen(get)-1]='\0';
              *nbytes = *nbytes+nsub-1;
            }
        }
        *ioerr=errno;
```

```
        return;
}
```

D Using the 'cu' command

To use the 'cu' command, the system manager must first edit the /etc/uucp/Devices file so that the baud rate for the serial port matches that being used by the 1440 instead of the default 9600 baud:

Direct ttyd1 - 4800 direct.

Then the following commands must be typed:

killall getty

telinit q.

To talk to the serial port /dev/ttyd1 connected to the 1440, type

cu -l ttyd1

and the console becomes a dummy terminal. You can now enter the same commands as you would with any other dummy terminal(i.e. commands that the 1440 understands). To exit the session and return to standard output, type '~.' .

It is not known how or why 'cu' chooses its set of terminal attributes.