

---

**HP 64700 Operating Environment**  
**Simulated I/O**

**User's Guide**



**HP Part No. B1471-97009**  
**Printed in U.S.A.**  
**November 1992**

**Edition 3**

---

## Notice

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.**

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

© Copyright 1991, 1992, Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and in other countries.

**Hewlett-Packard**  
**P.O. Box 2197**  
**1900 Garden of the Gods Road**  
**Colorado Springs, CO 80901-2197, U.S.A.**

**RESTRICTED RIGHTS LEGEND** Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

---

## Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1	B1471-97003, July 1991
Edition 2	B1471-97006, March 1992
<b>Edition 3</b>	<b>B1471-97009, November 1992</b>

## In This Book

---

This *Simulated I/O User's Guide* contains five chapters: an introduction to simulated I/O, two reference chapters, and one examples chapter. Also, there is a chapter which describes the error codes used with simulated I/O. A description of the five chapters is given below.

- Chapter 1** **Introducing Simulated I/O.** This chapter introduces you to the simulated I/O emulation feature by explaining what simulated I/O is, how it works, what steps you must take in order to use simulated I/O.
- Chapter 2** **Configuring Simulated I/O.** This chapter describes the simulated I/O configuration questions which appear as a part of the general emulation configuration questions. This chapter also lists restrictions on the use of simulated I/O.
- Chapter 3** **Simulated I/O Protocol.** This chapter describes how emulation system programs must communicate with the simulated I/O process in order to open, close, read from, and write to simulated I/O files/devices. This chapter also describes the communication process whereby emulation systems can execute UNIX processes.
- Chapter 4** **Examples.** This chapter shows you how to use the simulated I/O commands to use your workstation's display and keyboard as simulated I/O devices. This chapter also steps you through the process of compiling, linking, and running a simulated I/O demo program.
- Chapter 5** **Error Codes.** This chapter describes the simulated I/O error codes as they relate to the particular simulated I/O commands.

---

## Conventions

Example commands throughout the manual use the following conventions:

<b>bold</b>	Commands, options, and parts of command syntax.
<b><i>bold italic</i></b>	Commands, options, and parts of command syntax which may be entered by pressing softkeys.
normal	User specified parts of a command.
\$	Represents the UNIX prompt. Commands which follow the "\$" are entered at the UNIX prompt.
<RETURN>	The carriage return key.

# Contents

---

<b>1</b>	<b>Introducing Simulated I/O</b>	
	Overview . . . . .	9
	Simulated I/O Is ... . . . .	9
	How Does Simulated I/O Work? . . . . .	9
	User Program Request: . . . . .	10
	Response: . . . . .	11
	Using Simulated I/O . . . . .	11
<b>2</b>	<b>Configuring Simulated I/O</b>	
	Overview . . . . .	13
	Introduction . . . . .	13
	Answering The Simulated I/O Configuration Questions . . . . .	14
	Restrictions On Simulated I/O . . . . .	17
<b>3</b>	<b>Simulated I/O Protocol</b>	
	Overview . . . . .	19
	Introduction . . . . .	19
	Open File (90H) . . . . .	21
	Open File Option . . . . .	21
	Length of Path Name . . . . .	23
	Path Name . . . . .	23
	File Descriptor . . . . .	23
	Close File (91H) . . . . .	24
	File Descriptor . . . . .	24
	Read from File (92H) . . . . .	25
	File Descriptor . . . . .	26
	Number of Bytes to Read . . . . .	26
	Actual Number of Bytes Read . . . . .	26
	Actual Bytes Read . . . . .	26
	Write to File (93H) . . . . .	27
	File Descriptor . . . . .	27
	Number of Bytes to Write . . . . .	28
	Bytes to be Written . . . . .	28
	Actual Number of Bytes Written . . . . .	28

## Contents

Delete a File (94H) . . . . .	29
Path Name Length . . . . .	29
Path Name . . . . .	29
Position File Pointer (95H) . . . . .	30
Relative Byte Offset . . . . .	31
Starting Location . . . . .	31
Absolute Byte Offset . . . . .	32
Position Cursor on Display (96H) . . . . .	33
File Descriptor (Returned from Open) . . . . .	33
Line Number . . . . .	33
Column Number . . . . .	33
Clear Display (97H) . . . . .	34
File Descriptor (Returned from Open) . . . . .	34
UNIX Command (98H) . . . . .	35
Open Pipe Specification . . . . .	36
Command Name Length . . . . .	36
Command Name . . . . .	36
Simulated I/O Process ID . . . . .	36
Stdin File Descriptor . . . . .	36
Stdout File Descriptor . . . . .	37
Stderr File Descriptor . . . . .	37
Kill Simulated I/O Process (99H) . . . . .	38
Simulated I/O Process ID . . . . .	38
Signal to Send to Process . . . . .	38
Reset Simulated I/O (9AH) . . . . .	39

## 4 Examples

Overview . . . . .	41
Introduction . . . . .	41
Using Display Simulated I/O . . . . .	42
To Open Display Simulated I/O: . . . . .	42
To Write to Display Simulated I/O: . . . . .	43
To Position the Cursor: . . . . .	44
To Clear the Display: . . . . .	45
To Close Display Simulated I/O: . . . . .	46
Using Keyboard Simulated I/O . . . . .	46
To Open Keyboard Simulated I/O: . . . . .	46
Reading From the Keyboard: . . . . .	48
To Close the Keyboard Interface: . . . . .	49
Running The Simulated I/O Demo Program . . . . .	49
Copying the Simulated I/O Demo Program . . . . .	50

## Contents

Compiling the Simulated I/O Demo Program . . . . .	50
Copying the Default Emulator Configuration File . . . . .	50
Entering the Softkey Interface . . . . .	51
Configuring for Simulated I/O . . . . .	51
Loading the Absolute File . . . . .	52
Displaying Simulated I/O . . . . .	52
Running the Demo Program . . . . .	53
Modifying the Keyboard to Simulated I/O . . . . .	53
Using Emulation Commands while Simulated I/O is Running	55
Closing the Keyboard Simulated I/O Demo . . . . .	55
RS-232 Simulated I/O . . . . .	69
Configuring RS-232 Lines for Simulated I/O . . . . .	69
Serial Lines in UNIX Systems . . . . .	69
Serial Lines Used with Simulated I/O . . . . .	70
Serial Lines for User Terminals and Simulated I/O . . . . .	70
Dedicated Serial Lines . . . . .	71
Example Programs . . . . .	72
Special Considerations . . . . .	72

## 5 Error Codes

Overview . . . . .	79
Introduction . . . . .	79
Error Code Description (By Simulated I/O Command) . . . . .	81
General Errors . . . . .	81
Open (90H) Simulated I/O Errors . . . . .	81
Open File Errors . . . . .	82
Close (91H) Simulated I/O Errors . . . . .	82
Close File Errors . . . . .	82
Read (92H) Simulated I/O Errors . . . . .	82
Read File Errors . . . . .	82
Write (93H) Simulated I/O Errors . . . . .	83
Read File Errors . . . . .	83
Delete File (94H) Simulated I/O Errors . . . . .	83
Position File (95H) Simulated I/O Errors . . . . .	83
Position Cursor (96H) Simulated I/O Errors . . . . .	84
Clear Display (97H) Simulated I/O Errors . . . . .	84
UNIX Command (98H) Simulated I/O Errors . . . . .	84
Kill Process (99H) Simulated I/O Errors . . . . .	85





## Introducing Simulated I/O

---

### Overview

This chapter will answer the following questions:

- What is simulated I/O?
- Generally, how does simulated I/O work?
- What steps do I have to take to use simulated I/O?

### Note



---

Simulated I/O is not available when your emulator has been restricted to real-time runs.

---

---

### Simulated I/O Is ...

Simulated I/O is a process which allows your emulation system to communicate with host computer files, keyboard, and display.

Simulated I/O also allows your emulation system to execute UNIX commands; this means that, in addition to being able to communicate with your keyboard and display, your emulation system can also communicate with other host computer I/O devices, such as printers, plotters, modems, etc.

### How Does Simulated I/O Work?

Communication between your emulation system and host computer files takes place through contiguous byte length emulation system memory locations. The first memory location is called the Control Address (CA). The Control Address and the memory locations which follow it are referred to as the CA buffer.

Introducing Simulated I/O  
Simulated I/O Is ...



The Control Address (CA) buffer should be located in emulation RAM. It is possible, however, for the CA buffer to be located in target system RAM, but there will be a performance penalty for doing so. We recommend that you locate the Control Address and the CA buffer in emulation RAM.

Control Address buffers will never be more than 260 bytes in length because a maximum 256 bytes of information can be transferred between your emulation system and the host computer at a time. (Some simulated I/O commands require four additional bytes for command parameters.)

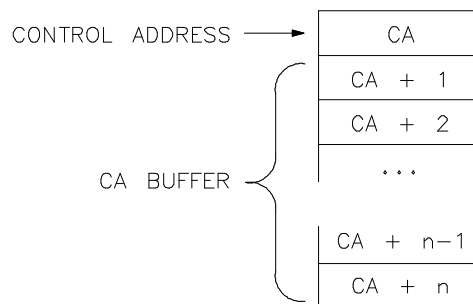
Communication between your emulation system and simulated I/O takes place as a series of requests by your emulation system and responses by simulated I/O.

**User Program Request:**

(1) Simulated I/O command parameters are first placed into the appropriate CA buffer locations.

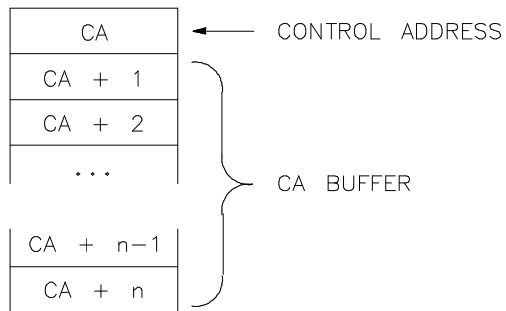
(2) After the CA buffer is loaded with simulated I/O command parameters, the command code is placed into the Control Address (CA) to cause the execution of the command.

USER PROGRAM REQUEST:



- Response:**
- (3) Before returning a value to the Control Address, some simulated I/O operations return additional information to the CA buffer.
  - (4) If the operation was successful, a 00H is returned to the Control Address. If the operation was not successful, an error code is returned to the CA.

RESPONSE:



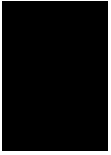
---

## Using Simulated I/O

The steps that you must take in order to use simulated I/O are listed below.

- You must write a program (to be executed by the emulator) which loads the CA buffer with the appropriate command parameters, places the appropriate command codes into the CA, and waits for the responses from simulated I/O. Command codes, command parameters, and simulated I/O responses are described in the "Simulated I/O Protocol" chapter.
- You must answer the simulated I/O emulation configuration questions. These questions are explained in the "Configuring for Simulated I/O" chapter.

The steps involved in using simulated I/O are also shown in the "Examples" chapter.



## Configuring Simulated I/O

---

### Overview

This chapter will:

- Explain the simulated I/O emulation configuration questions.
- List the restrictions on the use of simulated I/O.

---

### Introduction

The simulated I/O subsystem must be set up by answering a series of configuration questions. These questions are a part of the general emulation configuration. They deal with enabling simulated I/O, setting the control address(es), and defining files used for standard I/O. The simulated I/O configuration questions are reached by answering **yes** to the following question.

#### Note



---

Simulated I/O is not available if the emulator is configured for real-time runs. When the emulator is configured for real-time runs, the following questions will not appear among the emulator configuration questions.

---

## Answering The Simulated I/O Configuration Questions

### Modify simulated I/O configuration? **no** (yes)

*no* Answering no will cause the simulated I/O configuration questions to be skipped. The current simulated I/O configuration will not be modified.

*yes* Answering yes causes the following questions, which will allow you to modify the simulated I/O configuration, to be asked.

### Enable polling for simulated I/O?

*no* Prevents the emulation software from reading the control address for simulated I/O commands. Answering no to this question will allow you to disable simulated I/O while maintaining the current simulated I/O configuration. Later, when you wish to enable simulated I/O, you may do so without having to re-enter control addresses or the file names for standard input, standard output, and standard error output. Answering no will also cause the rest of the simulated I/O questions to be skipped.

*yes* Causes the emulation software to read the control address frequently to determine if the user program has requested any simulated I/O commands. Answering yes will also cause the following questions to be asked.

**Simulated I/O control address 1? SIMIO\_CA\_ONE**

**Simulated I/O control address 2? SIMIO\_CA\_TWO**

Configuring Simulated I/O

**Answering The Simulated I/O Configuration Questions**

**Simulated I/O control address 3? SIMIO\_CA\_THREE**

**Simulated I/O control address 4? SIMIO\_CA\_FOUR**

**Simulated I/O control address 5? SIMIO\_CA\_FIVE**

**Simulated I/O control address 6? SIMIO\_CA\_SIX**

The symbol SIMIO\_CA\_ONE is the default symbol associated with the first simulated I/O Control Address. The default symbol may be replaced with any other valid symbol or an absolute address. If a symbol is specified, polling on that control address will not begin until a file containing that symbol is loaded. If an absolute address is specified, polling on that address will begin immediately.

The control address must be loaded into memory space assigned as RAM. It is recommended that the control address be located in emulation RAM since this allows user programs to run faster. Using target RAM causes the emulator to break into the monitor program every time the control address is polled for simulated I/O commands or data.

The following questions deal with the files associated with the three reserved file names "stdin", "stdout", and "stderr".

**File used for standard input? /dev/simio/keyboard**

**File used for standard output? /dev/simio/display**

**File used for standard error output? /dev/simio/display**

The default answers for these questions are "/dev/simio/keyboard", "/dev/simio/display", and "/dev/simio/display" respectively.

These files are not actually opened until Open (90H) is called with the file names "stdin", "stdout", and "stderr" and are only provided to allow easy redirection of input and output from the keyboard or display to a file, etc., without modifying the user program. (The compiler standard I/O libraries may open some or all of these reserved files automatically if simulated I/O is used. For more details, see the documentation on the simulated I/O libraries for the compiler in question.)



Configuring Simulated I/O  
**Answering The Simulated I/O Configuration Questions**

**Enable simio status messages?**

- |            |  |
|------------|--|
| <i>yes</i> | The simulated I/O command and return code are displayed in the upper right hand corner of the display.       |
| <i>no</i>  | Disables the simulated I/O status messages. Simulated I/O runs faster when the status messages are disabled. |



**Note**

---

With HP 64700 firmware version 4.0 or greater, much of simulated I/O has been moved to the emulation card cage, and some simulated I/O operations complete without host intervention or notification (for example, reading from a file descriptor that has not been opened). With simulated I/O status messages enabled, there can be some simulated I/O commands that are not reported.

In addition, the simulated I/O write operation is now completed in the card cage, and any errors from write operations will not be reported on the write command that caused the error. Eventually, the host will report to the card cage that a write has failed and any write operations after this will receive the error.

---

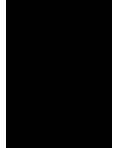


## **Restrictions On Simulated I/O**

The only two restrictions on the use of simulated I/O are:

- There is a limit of 12 open files at any one time.
- There may only be 4 active simulated I/O processes at any one time.

Since any simulated I/O file that is opened is associated with a file descriptor, opened files are independent of the Control Address. Up to 12 files may be opened with a single Control Address (CA). A total of 6 Control Addresses are allowed so that you can execute simulated I/O commands concurrently. Remember, a maximum of 12 simulated I/O files (between the 6 Control Addresses) may be opened at any one time.





## Simulated I/O Protocol

---

### Overview

This chapter will:

- Present the simulated I/O commands.
- Describe the simulated I/O command parameters.
- Describe the responses to simulated I/O commands.

---

### Introduction

Communication between your emulation system and simulated I/O takes place as a series of requests by your emulation system to execute simulated I/O commands and responses by simulated I/O which tell your emulation system whether the command was successful or not. The communication between your emulation system and simulated I/O is done through byte length emulation system RAM locations (preferably emulation RAM, which is faster, but possibly target system RAM with less performance). All simulated I/O command codes and parameters are placed into memory locations relative to a Control Address (CA) memory location. (Control Addresses are assigned when configuring your emulation system for simulated I/O.) There are 10 simulated I/O commands:

Simulated I/O Protocol  
**Introduction**

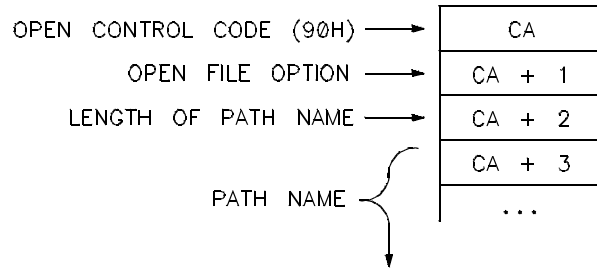
<b>Command Code</b>	<b>Simulated I/O Command</b>
90H	Open file
91H	Close file
92H	Read from file
93H	Write to file
94H	Delete a file
95H	Position file pointer
96H	Position cursor (display simulated I/O only).
97H	Clear display (display simulated I/O only).
98H	UNIX command
99H	Kill simulated I/O process.
9AH	Reset simulated I/O.

This chapter contains descriptions of the simulated I/O protocol, that is, the rules which govern the exchange of messages between your emulation system and simulated I/O.

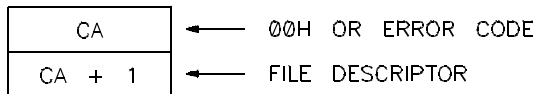
## Open File (90H)

Open (90H) opens a file or device for reading and/or writing by simulated I/O.

REQUEST TO OPEN FILE/DEVICE:



RESPONSE:



To invoke open, an open file option must be stored in address CA + 1, the length of the path name must be stored in location CA + 2, and the path name must be stored in CA + 3 and the following locations. Then, the open control code (90H) is stored in CA.

### Note



The maximum number of simulated I/O files that may be open at any one time is 12.

### Open File Option

The open file option stored in location CA + 1 must be one or more of the following flags OR-ed together:

**Note**



---

Exactly one of the first three open option flags must be used.

---

Read Only (code = 00H)

Write Only (code = 01H)

Read and Write (code = 02H)

Create (code = 04H) The file is opened if it exists; otherwise, a new file is created and opened.

Append (code = 08H) The current position is changed to the end of file prior to each write.

Truncate (code = 10H) If the file exists, its length is truncated to 0.

Exclusive (code = 20H) If both this flag and the create flag are set, open will fail if the file exists. It has no effect if the create flag is not set.

No Delay (code = 40H) Read requests on this file will return immediately. With ordinary files this has no effect, but with devices such as a tty, emulation will not wait until data is available. This flag may affect subsequent reads and writes. See open (2), read (2), and write (2) in the UNIX Reference manual for more details.

Open will fail if the file does not exist unless the create option is set. The open will succeed if the create option is set even if the file already exists, unless the exclusive option is also set. Upon opening, the file position pointer is set to the beginning of the file unless the append option is used where it points to the end of the file.

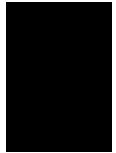
**Length of Path Name** The length of the path name placed into location CA + 2 is the number of bytes in the path name where each (ASCII) character represents one byte. The "path name" can be shorter than the "length of path name" if the path name is null terminated (null = 00H). The path name need not be null terminated if its length is exact. (A null will be added after the specified number of characters.)

**Path Name** The files "stdin", "stdout", and "stderr" are reserved for simulated I/O. When files with these names are opened, the file specified at configuration time for the standard I/O name requested will be the file actually opened. NOTE: unlike commands executed under a UNIX shell, these files are NOT automatically opened, but must be opened by your program.

The file names "/dev/simio/keyboard" and "/dev/simio/display" are also reserved. They represent devices for simulated I/O keyboards and displays. These file names are interpreted internally by the emulation software and do not exist as actual UNIX files. See examples of display and keyboard simulated I/O in the "Examples" chapter.

Absolute path names must be specified for the file names reserved for simulated I/O. Relative path names, as well as absolute path names, may be specified for other simulated I/O files.

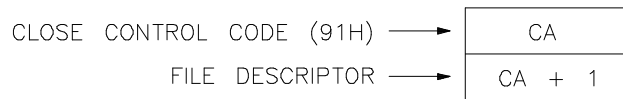
**File Descriptor** The file descriptor returned by **open** should be saved for use by all reads, writes, seeks, or closes on the opened file. All simulated I/O commands which require a file descriptor specify that the file descriptor be in location CA + 1, and none of these commands modify the location CA + 1. Therefore, the file descriptor returned by open need not be saved if a unique Control Address is used for each open file or device, and the value at location CA + 1 is never modified from the time the file is opened until the file is closed. The file descriptor is one created and managed by emulation software and is not an actual UNIX file descriptor.



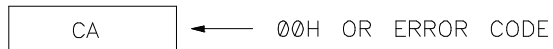
### Close File (91H)

Close (91H) closes the file or device associated with the file descriptor specified. The file descriptor of the desired file is stored in CA + 1, and 91H is then stored in CA to invoke the close command. Zero is returned in the CA if the operation was successful, otherwise an error code is returned indicating why it failed.

REQUEST TO CLOSE FILE/DEVICE:



RESPONSE:



### File Descriptor

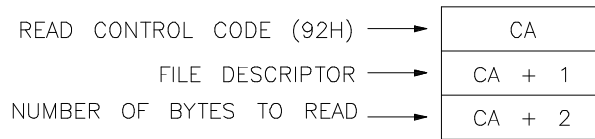
The file descriptor placed into CA + 1 is the file descriptor that was returned by simulated I/O when the file was originally opened.



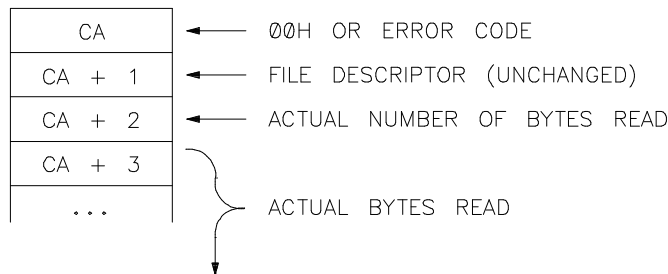
## Read from File (92H)

To initiate a read operation, first store the file descriptor of the open file into CA + 1. Then, store the number of bytes (maximum of 255) to read in location CA + 2. A buffer large enough for the bytes requested must be provided starting at location CA + 3. Finally, store the read control code (92H) into location CA to initiate the read.

REQUEST TO READ FILE/DEVICE:



RESPONSE:



When the read operation is complete, the number of bytes actually read is returned to location CA + 2, the bytes read are stored in the buffer starting at location CA + 3, and 00H is returned to location CA if the operation was successful, or a nonzero error code is returned to location CA indicating why the operation failed.

Read behaves differently for the simulated keyboard device. The read will return only after a line has been completed (a < RETURN> or one of the softkeys has been pressed). The read will never return more than a single line of input. (A single line will return unless the number of characters requested is smaller than the length of the input; multiple reads will be required in this case, but a single read will never return text from more than one line, even if more than one line of text is available.) This will

Simulated I/O Protocol  
**Read from File (92H)**

generally be less than the requested number of characters. See the keyboard simulated I/O example in the "Examples" chapter.

**File Descriptor**

The file descriptor placed into CA + 1 is the file descriptor that was returned by simulated I/O when the file was originally opened.

**Number of Bytes to Read**

The number of bytes to read is placed in location CA + 2. The maximum number of bytes that can be read per read command is 255. Make sure that the CA buffer is big enough to hold the number of bytes that you wish to read.

**Actual Number of Bytes Read**

The end of file is indicated when 00H is returned to CA + 2. No error is indicated if fewer than the specified number of bytes are actually read. (This can happen when there are fewer than the specified number of bytes remaining in the file and, in most cases, when reading from the keyboard.)

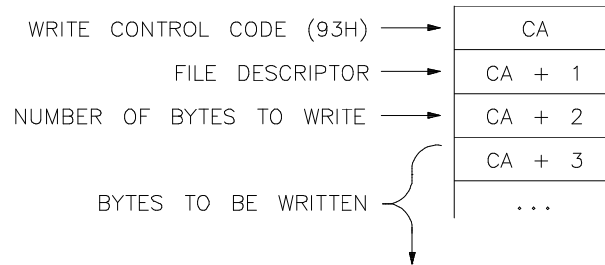
**Actual Bytes Read**

The bytes that were read from the file/device are placed into CA buffer locations starting at location CA + 3. A maximum number of 255 bytes can be read.

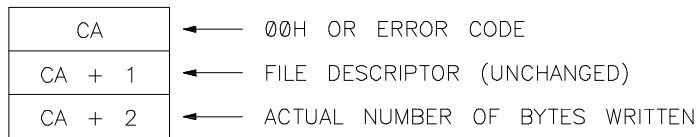
## Write to File (93H)

To initiate a **write** operation, a buffer containing the bytes to be written must be set up at location CA + 3. The actual number of bytes to write (maximum of 255) must be stored in location CA + 2, and the file descriptor of an open file must be stored in location CA + 1. Finally, the **write** control code (93H) must be stored in location CA.

REQUEST TO WRITE TO FILE/DEVICE:



RESPONSE:



When the **write** operation is complete, 00H is returned to location CA if the write was successful, and the actual number of bytes written is returned to location CA + 2 (the file descriptor in CA + 1 is unchanged); otherwise, an error code is returned to location CA indicating why the operation failed, and the number of bytes written is set to 00H.

If the file was opened with the "append" option, the file pointer is positioned to the end of file prior to each write. (NOTE: the "append" option is ignored if the file is the display.)

### File Descriptor

The file descriptor placed into CA + 1 is the file descriptor that was returned by simulated I/O when the file was originally opened.

Simulated I/O Protocol  
**Write to File (93H)**

**Number of Bytes to Write**

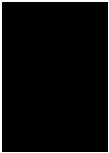
When requesting to write, the number of bytes to write is placed into location CA + 2. The maximum number of bytes that can be written per write command is 255. Make sure that the CA buffer is big enough to hold the number of bytes you wish to write.

**Bytes to be Written**

The bytes to be written are placed into the CA buffer starting at location CA + 3. The maximum number of bytes that can be written is 255.

**Actual Number of Bytes Written**

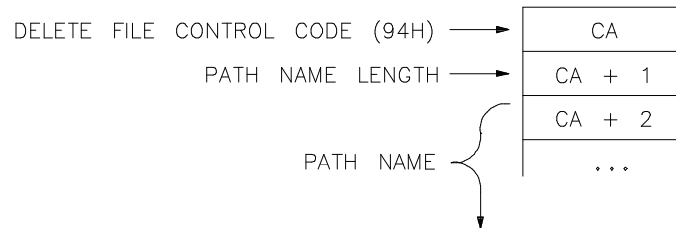
If the write operation was successful, simulated I/O will tell you how many bytes were actually written by placing their number into location CA + 2.



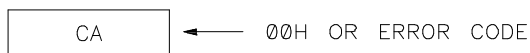
## Delete a File (94H)

Delete File (94H) removes the directory entry named. This action is invoked by storing the length of the file name in location CA + 1, the file name (relative path or absolute path) starting at location CA + 2, and then storing the delete file control code (94H) into the CA.

REQUEST TO DELETE A UNIX FILE:



RESPONSE:



### Path Name Length

The length of the path name placed into location CA + 1 is the number of bytes in the path name where each (ASCII) character represents one byte. The "path name" can be shorter than the "path name length" if the path name is null terminated (null = 00H). The path name need not be null terminated if its length is exact. (A null will be added after the specified number of characters.)

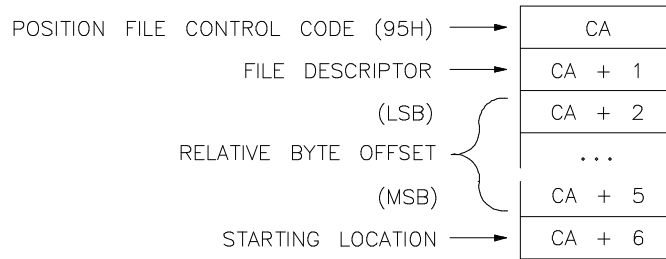
### Path Name

Note that the file names for delete file are not interpreted by simulated I/O; therefore, special files such as "stdin" or "/dev/simio/display" are treated as standard UNIX files. For example, trying to delete a file called "stdin" will attempt to to remove the file "./stdin", not the special simulated I/O "stdin" file.

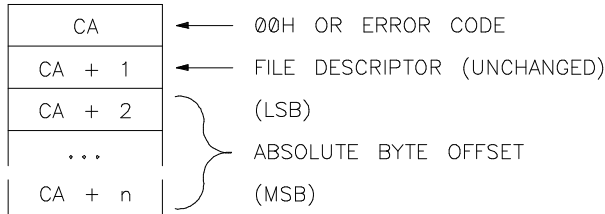
## Position File Pointer (95H)

To initiate a **position file** operation, store the file descriptor into CA + 1, the starting code (see below) into location CA + 6, and a 32 bit **SIGNED** integer into locations CA + 2 through CA + 5. The 32 bit signed integer will indicate a byte offset from the starting code. The least significant byte of the offset will be in location CA + 2, and the most significant byte will be in location CA + 5. After the file descriptor, the byte offset, and the starting code are placed in the CA buffer, store the position file control code (95H) into CA.

REQUEST TO CHANGE FILE POSITION POINTER:



RESPONSE:



When the operation is complete, the absolute offset from the beginning of the file is returned into locations CA + 2 through CA + 5 (a 32 bit **SIGNED** integer whose least significant byte is in location CA + 2 and whose most significant byte is in location CA + 5), and 00H is returned to CA if the operation was successful or an error code is returned to CA if the operation failed.

**Note**



---

If the file was opened with the "append" option, the position pointer will automatically be repositioned to the end of file before any write operation, even if you use a position command before the write. You can, however, use the position file command followed by a read command to read from a specific position in a simulated I/O file that was opened with the "append" option.

---

This command has no effect on the special simulated I/O files "/dev/simio/keyboard" and "/dev/simio/display", and if it is used, an error code for INVALID COMMAND (09H) will be returned.

FILE DESCRIPTOR. The file descriptor placed into CA + 1 is the file descriptor that was returned by simulated I/O when the file was originally opened.

**Relative Byte Offset**

The relative byte offset is a 32 bit SIGNED integer which indicates a byte offset from the starting location. The relative byte offset is placed into locations CA + 2 through CA + 5 (with the least significant byte in location CA + 2 and the most significant byte in location CA + 5).

**Starting Location**

The starting location placed in location CA + 6 will determine the position to which the "relative byte offset" will be added to determine the resulting pointer position. The codes which may be placed in the "starting location" are:

starting code = 00H      The offset is from the beginning of the file.

starting code = 01H      The offset is from the current position.

starting code = 02H      The offset is from the end of the file.

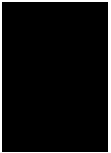
To **rewind** a file, use starting code = 00H and offset = 00H. To **find the current position** in the file, use the starting code = 01H and offset = 00H (the current location is returned in the "absolute offset" from the beginning of the file).



Simulated I/O Protocol  
**Position File Pointer (95H)**

**Absolute Byte Offset**

The absolute byte offset returned to locations CA + 2 through CA + 5 is a 32 bit SIGNED integer whose least significant byte is in location CA + 2 and whose most significant byte is in location CA + 5.

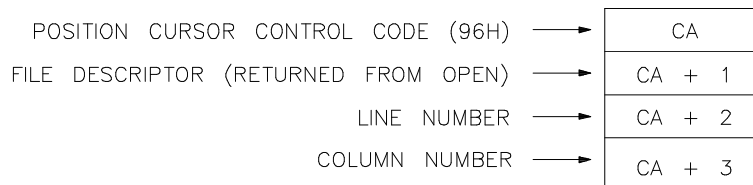




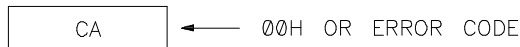
**Position Cursor on Display (96H)**

To position the cursor for writes to the screen, store the file descriptor returned from the open command in location CA + 1, the desired line number (0 through 49) in location CA + 2, and the desired column number (0 through 79 or one minus the number of columns on the display device if greater than 80) into location CA + 3. Then, store the position cursor control code (96H) into CA. The next write will begin at this location. Zero is returned to location CA if the position cursor operation is successful; otherwise, a nonzero error code is returned to CA.

REQUEST TO POSITION CURSOR:



RESPONSE:



**File Descriptor (Returned from Open)**

The file descriptor that is placed into location CA + 1 is the file descriptor that was returned when the file was originally opened.

**Line Number**

The line number placed into location CA + 2 must be one of the 50 lines of display simulated I/O (0 through 49).

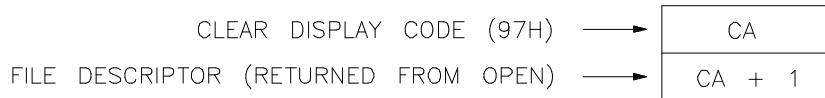
**Column Number**

The column number placed into location CA + 3 must be 0 through 79 (or one minus the number of columns on the display device if greater than 80).

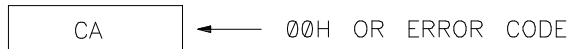
## Clear Display (97H)

To clear the display, the file descriptor returned from open must be stored into location CA + 1, the clear display control code (97H) is stored into location CA. Zero is returned to location CA if the operation was successful; otherwise, a nonzero error code is returned to CA. When the display is cleared, the cursor is left at column 0, line 0.

REQUEST TO CLEAR THE DISPLAY:



RESPONSE:



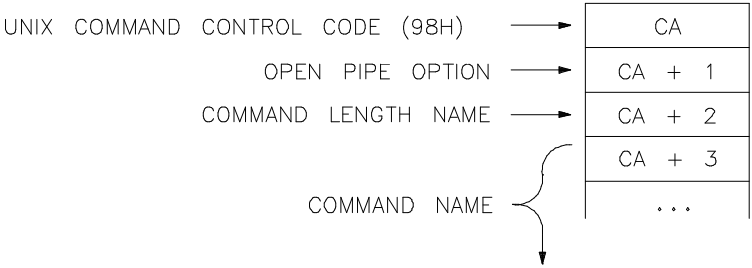
### File Descriptor (Returned from Open)

The file descriptor that is placed into location CA + 1 is the file descriptor that was returned when the file was originally opened.

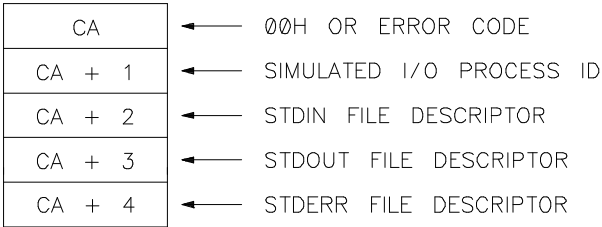
**UNIX Command (98H)**

UNIX Command (98H) allows you to execute a UNIX command from emulation. In order to execute a UNIX command, emulation forks off a process which then opens pipes to the first three file descriptors (the requested command's standard in, out, and error). This forked process will then execute the requested command in a sub shell with communication (if requested) to the shell's "stdio".

REQUEST TO EXECUTE UNIX COMMAND:



RESPONSE:



**Note** 

A maximum number of 4 simulated I/O processes can be active at any one time.

If any pipes are opened to the simulated I/O process (i.e., bits in the Open Pipe Option are SET), they are considered to be open files. A maximum of 12 open files are allowed at any one time.

### Open Pipe Specification

The pipes to open are specified as a bitmap in location CA + 1.

- Bit 0 set specified "stdin" to be opened.
- Bit 1 set specifies "stdout" to be opened.
- Bit 2 set specifies "stderr" to be opened.

If any of the three bits is cleared, "/dev/null" will be opened. Inputs to the UNIX command can be sent by using the Write (93H) command and the "stdin" file descriptor which is returned to location CA + 2. Likewise, outputs from the UNIX command can be read by using the Read (92H) command and the "stdout" and "stderr" file descriptors which are returned to locations CA + 3 and CA + 4, respectively.

### Command Name Length

The length of the command name placed into location CA + 2 is the number of bytes in the command name where each (ASCII) character represents one byte. The "command name" can be shorter than the "command name length" if the name is null terminated (null = 00H). The command name need not be null terminated if its length is exact. (A null will be added after the specified number of characters.)

### Command Name

The command name is executed as an argument to "sh" executed with the "-c" option so any valid shell command line can be specified and shell features such as I/O redirection, pipes, and filename expansion using "\*", "?", etc., can be used. Thus, the strings like "pr | lpr" and "ls \*.c > source\_files" are valid command names.

### Simulated I/O Process ID

The "simulated I/O process ID" returned to location CA + 1 is a process ID internal to simulated I/O and, therefore, does not directly correspond to a UNIX process ID. This allows the ID to be contained in one byte.

### Stdin File Descriptor

If the open pipe option specifies that "stdin" be opened, the descriptor of the file that is opened is returned to location CA + 2.

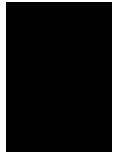
Any inputs to the UNIX command may be written to the "stdin" file by using the simulated I/O Write (93H) command, just as any other simulated I/O file is written to.

**Stdout File Descriptor**

If the open pipe option specifies that "stdout" be opened, the descriptor of the file that is opened is returned to location CA + 3. Any outputs from the UNIX command can be read from the "stdout" file by using the simulated I/O Read (92H) command, just as any other simulated I/O file is read from.

**Stderr File Descriptor**

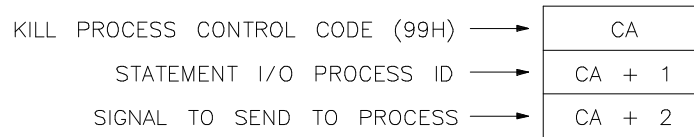
If the open pipe option specifies that "stderr" be opened, the descriptor of the file that is opened is returned to location CA + 4. Any error outputs from the UNIX command can be read from the "stderr" file by using the simulated I/O Read (92H) command, just as any other simulated I/O file is read from.



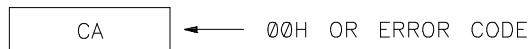
## **Kill Simulated I/O Process (99H)**

Kill (99H) can be called to terminate the execution of a UNIX process started under simulated I/O. This command sends the signal specified in CA + 2 to the requested process.

REQUEST TO KILL SIMULATED I/O PROCESS:



RESPONSE:



This command can also be used to determine if a process is still running by sending a signal 00H to the process. A return value of 00H indicates that the process is still active and a nonzero return value indicates that the process no longer exists.

### **Simulated I/O Process ID**

The simulated I/O process ID placed into location CA + 1 is the process ID that was returned from the simulated I/O UNIX Command (98H) operation.

### **Signal to Send to Process**

See **signal (2)** in the UNIX reference manual for information on the signal to send to the process.

## **Reset Simulated I/O (9AH)**

Reset simulated I/O (9AH) closes all open simulated I/O files and removes all entries from the simulated I/O process table. The processes are not explicitly killed, but SIGPIPE is sent to each process connected by a pipe to simulated I/O when the pipe is closed; this, by default, kills the process. This command is provided to allow restarting a program running on an emulated processor without having side effects from previously opened files or previously executed processes.

REQUEST TO RESET SIMULATED I/O PROCESS:

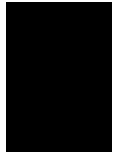
RESET SIMULATED I/O CONTROL CODE (9AH) → 

CA
----

RESPONSE:

CA
----

 ← 00H OR ERROR CODE







## Examples

---

### Overview

This chapter will:

- Show you how to use display simulated I/O.
- Show you how to use keyboard simulated I/O.
- Step you through the process of running the simulated I/O demo program.

---

### Introduction

Simulated I/O lets you use your workstation display and keyboard as emulation system output and input devices. This chapter will show you how to use the simulated I/O commands for display and keyboard simulated I/O.

This chapter also contains a "C" demo program which uses display and keyboard simulated I/O, and other simulated I/O commands as well. The demo program in this chapter has been included with the Softkey Interface emulation software. The examples at the end of this chapter step you through the process of compiling, linking, and running the demo program in emulation.

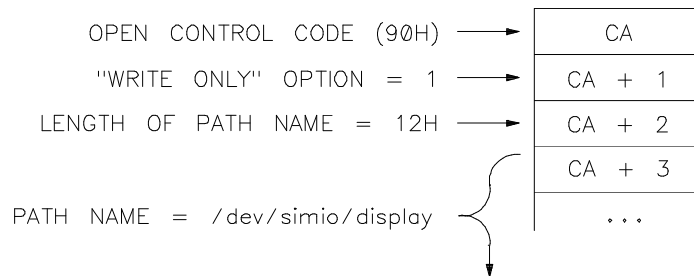
## Using Display Simulated I/O

The simulated I/O display interface allows your emulation system program to access the workstation display. The use of the simulated display does not actually put anything on the display until activated by pressing the "display" then "sim\_io" softkeys. Instead, the characters are stored in a buffer and displayed when the simulated I/O display is activated. This allows you to use the emulation "display" command to display memory, registers, etc., while your display simulated I/O program is running. The display buffer can always be updated (by a write or clear) whether it is currently being displayed or not. No data is lost if the workstation display is being used for other emulation commands while your simulated program is writing to display simulated I/O.

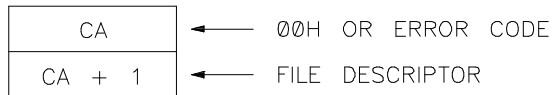
### To Open Display Simulated I/O:

To open the display, the standard simulated I/O open command is used. The file descriptor returned by "open" must be saved for all accesses to the display. The special file name "/dev/simio/display" is reserved for use by the simulated I/O display. This file does not correspond to any actual UNIX file, but instead is internal to the emulation software.

TO OPEN DISPLAY SIMULATED I/O:



RESPONSE:



The display opened is a 50 line by at least 80 column buffer. (The actual number of columns is 80 or the number of columns on the display device, whichever is greater.) The display buffer has associated with it a current cursor location, and all writes begin at this current cursor location. When the display is open, the cursor is located at the upper left corner of the display buffer (row 0, column 0), and the cursor position is updated after each write and position cursor command.

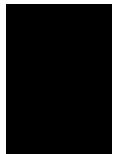
Only 16 lines of the buffer can be displayed at one time on the standard 24 line display; the actual number of lines displayed will be the number of lines on the display device minus 2 lines for the simulated I/O header and 6 lines for the status, command and softkey lines.

The < NEXT PAGE> , < PREV PAGE> , < ROLL UP> , and < ROLL DOWN> keys can be used to position the display anywhere in the buffer when the simulated I/O display is the active emulation display. (But, note that the display is updated to include the current cursor position in the visible region after each **write** or **position cursor** command.)

Any write past the last column will automatically wrap to the next line, and any write past the last column on the 50th line will cause all the lines to be scrolled up by one.

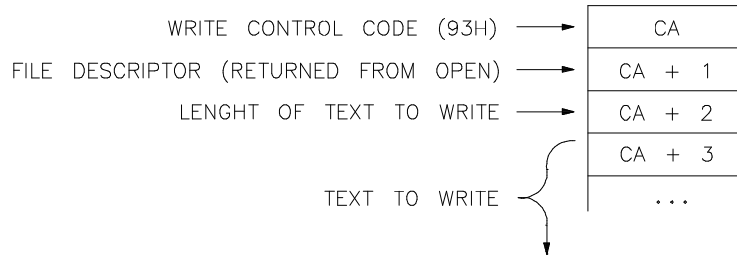
### To Write to Display Simulated I/O:

To write to the display, the standard simulated I/O write command is used. The file descriptor returned from the open command must be stored in location CA + 1. A buffer containing the text to be written must be set up starting at location CA + 3. The length of the text must be stored in location CA + 2; then, the write control code (93H) must be written to location CA. Zero is returned to the control address, and the number of bytes actually written is returned to location CA + 2 if the write was successful; a nonzero error code is returned to location CA if the write operation was not successful.

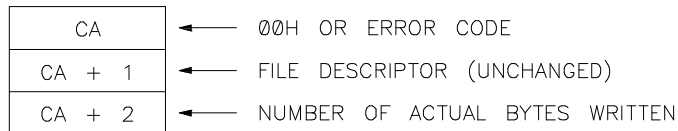


Examples  
Using Display Simulated I/O

TO WRITE TO DISPLAY SIMULATED I/O:



RESPONSE:



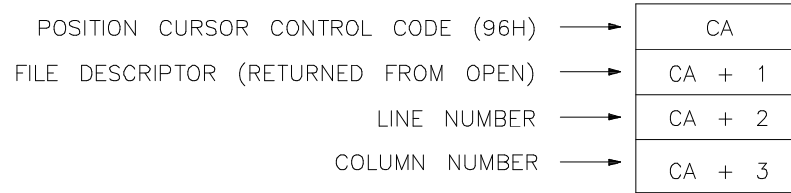
If the text to be written spans more than one line (contains more characters than the number of columns or contains a newline character), the writing will continue on the next line at column 0.

Text is written to the current cursor position, and the cursor position is updated after each write.

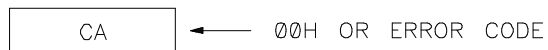
**To Position the  
Cursor:**

To position the cursor for writes to the screen, store the file descriptor returned from the open command in location CA + 1, the desired line number (0 through 49) in location CA + 2, and the desired column number (0 through 79 or one minus the number of columns on the display device if greater than 80) in location CA + 3. Then, store the position cursor command code (96H) in location CA. The next write will begin at this location. Zero is returned to location CA if the position command is successful; a nonzero error code is returned otherwise.

TO POSITION THE CURSOR:



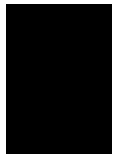
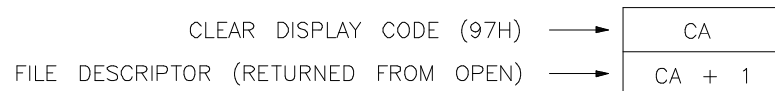
RESPONSE:



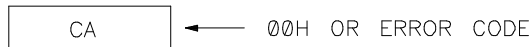
**To Clear the Display:** To clear the display, the file descriptor returned from open must be stored in location CA + 1, the clear display command code (97H) is stored in location CA. Zero is returned to location CA if the command was successful; otherwise, a nonzero error code is returned to location CA.

The display is cleared, and the cursor is left at row 0, column 0.

TO CLEAR THE DISPLAY:



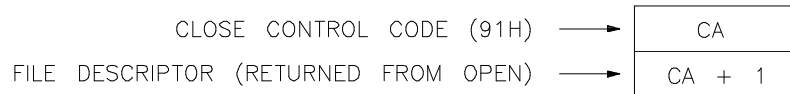
RESPONSE:



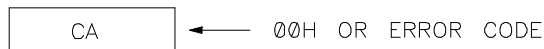
Examples  
**Using Keyboard Simulated I/O**

**To Close Display Simulated I/O:** To close the display, the standard simulated I/O close command is used. The file descriptor returned from the open command must be placed in location CA + 1; then, the close command code (91H) is placed into location CA. Zero is returned to CA if the close is successful; otherwise, a nonzero error code is returned.

TO CLOSE DISPLAY SIMULATED I/O:



RESPONSE:



---

**Using Keyboard Simulated I/O**

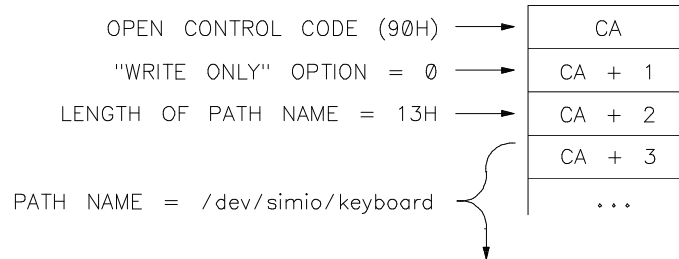
The simulated I/O keyboard interface allows your emulation system program to access the workstation keyboard as an input device.

**To Open Keyboard Simulated I/O:** To open the simulated I/O keyboard, use the standard simulated I/O open command. The open option code stored in location CA + 1 should be 0 for read only. Simulated I/O uses the special file name "/dev/simio/keyboard" to indicate the keyboard interface; this name must be stored in the CA buffer starting at location CA + 3, and the length of the name (13H) must be stored in location CA + 2. The open file command code must be stored in location CA. Zero is returned to location CA if the open is successful; otherwise a nonzero error code is returned. A file descriptor is

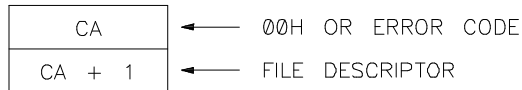
Examples  
**Using Keyboard Simulated I/O**

returned by the open command to location CA + 1 if the open is successful.

TO OPEN KEYBOARD SIMULATED I/O:



RESPONSE:



This file descriptor must be saved for use by all other commands accessing the keyboard.

Opening the keyboard does not actually direct keyboard characters to simulated I/O until you activate the simulated I/O keyboard by pressing the "modify" softkey followed by the "keyboard\_to\_simio" softkey. Modifying the keyboard to simulated I/O clears the command line area of the workstation display and begins storing the keyed in lines for reading by the simulated I/O system.

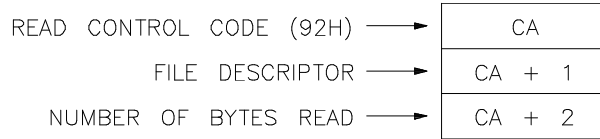
All characters input from the keyboard are displayed in the command line area of the display as they are being typed. The command line can be edited using the standard command line editing facilities before < RETURN> is pressed. The input line can be terminated by pressing < RETURN> , any of the softkeys, or < CTRL> **d**. No characters can be read by the simulated I/O program until < RETURN> , one of the softkeys, or < CTRL> **d** is pressed.



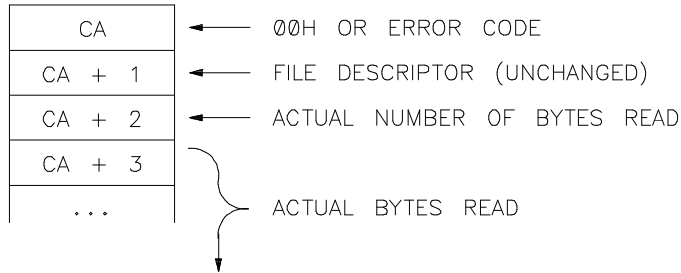
Examples  
Using Keyboard Simulated I/O

**Reading From the Keyboard:** To read from the keyboard, the standard simulated I/O read command is used.

READING FROM THE KEYBOARD:



RESPONSE:



The keyboard input is saved as a list of lines, and each call to read will read a maximum of one line of text. (The actual number of characters read will be returned.) The last characters of the line read will be a newline, which indicates that the input line was terminated by a < RETURN> or an escape character and an ASCII 1 through 8, indicating one of the softkeys f1 through f8 was used to terminate the input line. (Note that f1 corresponds to the "suspend" softkey.) If the keyboard input line was terminated with a < CTRL> **d**, no characters are added to the line. If the number of characters to read is less than the length of the line, multiple calls to read will be required to read the complete line, and no characters will be lost. The maximum length of a keyboard input line is 240 characters; therefore, a complete line will always be read if the number of characters requested in the read command is 240.

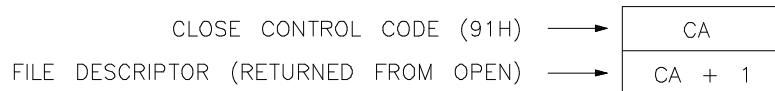


**Running The Simulated I/O Demo Program**

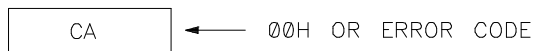
**To Close the Keyboard Interface:**

To close the keyboard interface, the standard simulated I/O close command is used. The file descriptor returned from the open command must be placed in location CA + 1; then, the close command code (91H) is placed into location CA. Zero is returned to CA if the close is successful; otherwise, a nonzero error code is returned.

TO CLOSE THE KEYBOARD:



RESPONSE:




---

**Running The Simulated I/O Demo Program**

The simulated I/O demo program which follows is written in the "C" language. This section will step you through the process of compiling and linking the demo program and running it in emulation.

The examples that follow have been created in the HP 64742 68000 emulation environment. The choice to run the demo program in the HP 64742 emulation environment is purely arbitrary, and most of the instructions that follow will be very similar, if not the same, for whichever emulation system you choose run this demo in.

## Examples

### Running The Simulated I/O Demo Program

#### Copying the Simulated I/O Demo Program

The simulated I/O demo program shown in figure 4-3 has been included with your emulation software and is located in the directory `/usr/hp64000/demo/emul/simio`. To copy the demo program to your directory, enter the commands shown below.

(The period just before the `<RETURN>` specifies that the file will have the same name in your current directory.)

```
$ cp /usr/hp64000/demo/emul/simio/simiodemo.h .  
<RETURN>  
$ cp /usr/hp64000/demo/emul/simio/simiodemo.c .  
<RETURN>
```

#### Compiling the Simulated I/O Demo Program

To compile the `simiodemo.c` demo program with the AxLS (Advanced Cross Language System) 68000 C Cross Compiler, enter the following command.

```
$ cc68000 -vOGr hp64742 -o simiodemo  
simiodemo.c <RETURN>
```

(There is one duplicate symbol error when compiling the program this way, but the compiler ignores the symbol that should be ignored, and the generated absolute file works correctly.)

Notice that the compiler's "-N" (no I/O) option is used. This is because the demo program contains routines that use simulated I/O. When the "-N" option is not used, the compiler links in its own I/O routines which use simulated I/O.

#### Copying the Default Emulator Configuration File

Since the AxLS 68000 C Cross Compiler provides default configuration files for the HP 64742 68000 Emulator, copy the default emulator configuration file to the current directory before you enter the emulation system.

```
$ cp /usr/hp64000/env/hp64742/config.EA  
config.EA <RETURN>
```

**Running The Simulated I/O Demo Program****Entering the Softkey Interface**

If you have installed your emulator and Softkey Interface software, opened a window, and set and exported the proper environment variables as directed in the *Softkey Interface Installation Notice*, you can enter the Softkey Interface with the following command:

```
$ emul1700 <emul_name> <RETURN>
```

The "emul\_name" in the command above is the logical emulator name given in the HP 64700 emulator device table (/usr/hp64000/etc/64700tab.net).

**Configuring for Simulated I/O**

First, load the compiler's emulator configuration file.

```
load configuration config <RETURN>
```

Now, modify the configuration to enable the simulated I/O feature and to give the Softkey Interface the demo program's control address.

```
modify configuration <RETURN>
```

Now step through the emulation configuration questions, by pressing the < RETURN> key, until you come to the question:

```
Modify simulated I/O configuration? yes
<RETURN>
```

Answer the simulated I/O configuration questions as shown below.

```
Enable polling for simulated I/O? yes
<RETURN>
Simulated I/O control address 1?
SIMIO_CA_ONE control_addr <RETURN>
Simulated I/O control address 2?
SIMIO_CA_TWO <RETURN>
Simulated I/O control address 3?
SIMIO_CA_THREE <RETURN>
Simulated I/O control address 4?
SIMIO_CA_FOUR <RETURN>
Simulated I/O control address 5?
SIMIO_CA_FIVE <RETURN>
Simulated I/O control address 6?
SIMIO_CA_SIX <RETURN>
```



## Examples

### Running The Simulated I/O Demo Program

```
File used for standard input?  
/dev/simio/keyboard <RETURN>  
File used for standard output?  
/dev/simio/display <RETURN>  
File used for standard error output?  
/dev/simio/display <RETURN>  
Enable simio status messages? yes
```

Press < RETURN> for the rest of the emulation configuration questions until you are asked the name of the configuration command file. Answer as shown below.

```
Command file name? simiodemocfg <RETURN>
```

Simulated I/O is now configured for the demo program. Notice that the demo program only uses one CA. The name "control\_addr" is the symbol associated with the "control\_addr" array defined in the **simiodemo.c** program.

### Loading the Absolute File

To load the absolute file, enter the following command:

```
load simiodemo.x <RETURN>
```

### Displaying Simulated I/O

To display the contents of the display simulated I/O buffer, enter the command shown below.

```
display simulated_io <RETURN>
```

The simulated I/O display is shown in figure 4-1.

```
Simulated I/O display

STATUS: M68000--Running in monitor_____...R....
_display simulated_io

run      trace      step      display      modify      break      end      ---ETC--
```

Figure 4-1. Displaying Simulated I/O

### Running the Demo Program

To run the simulated I/O demo program, enter the following command:

```
run from entry <RETURN>
```

The symbol "entry" was created by the "C" library which was linked with the demo program. The message "KEYBOARD NOW OPEN" will be written to the display simulated I/O buffer and will be displayed on the screen.

### Modifying the Keyboard to Simulated I/O

To be able to use your workstation keyboard as the standard input to the simulated I/O demo program, you must first activate your keyboard with the command shown below.

```
modify keyboard_to_simio <RETURN>
```

Your keyboard is now active as the simulated I/O input device as you can tell by the appearance of the suspend softkey. You can now



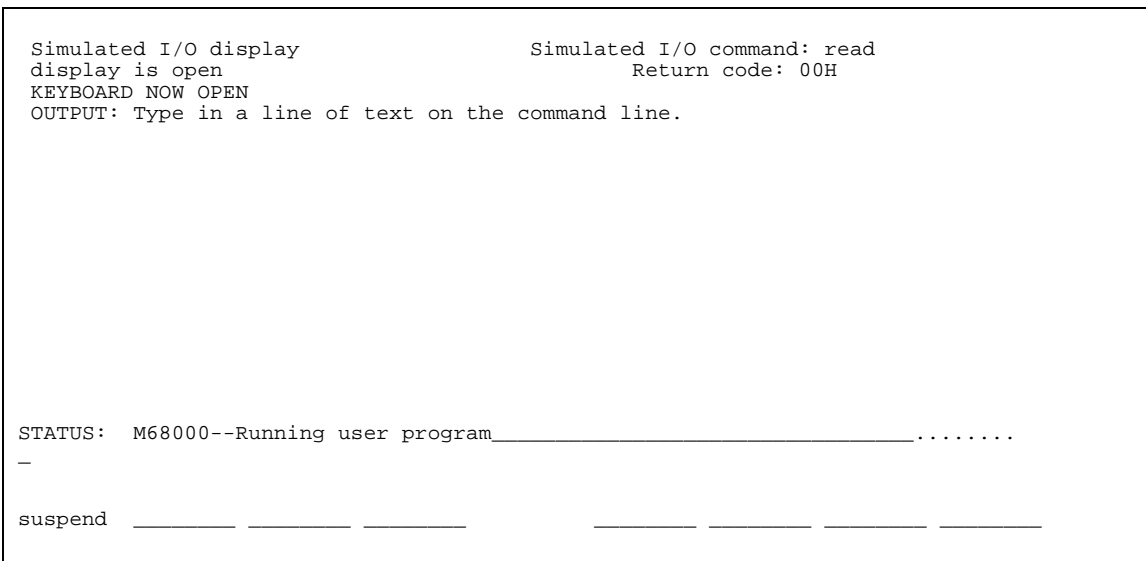
## Examples

### Running The Simulated I/O Demo Program

enter text on the command line using the standard command line editing features.

Type in a line of text on the command line.  
<RETURN>

The text that you just entered will appear on the simulated I/O display (see figure 4-2). Reads of the simulated I/O keyboard will only take place after the < RETURN> key or one of the softkeys has been pressed.



```
Simulated I/O display          Simulated I/O command: read
display is open                Return code: 00H
KEYBOARD NOW OPEN
OUTPUT: Type in a line of text on the command line.

STATUS: M68000--Running user program_____
-
suspend _____
```

**Figure 4-2. Entering Text from the Keyboard**

Pressing the "suspend" softkey will not close the simulated I/O keyboard; it will, however, deactivate your workstation keyboard as the simulated I/O input device, and allow you enter emulation commands.

## Running The Simulated I/O Demo Program

### Using Emulation Commands while Simulated I/O is Running

It is possible to enter emulation commands while your simulated I/O program is running. For example, enter the commands shown below.

```
suspend
trace <RETURN>
display trace <RETURN>
display simulated_io <RETURN>
modify keyboard_to_simio <RETURN>
```

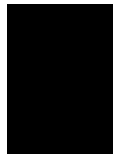
You once again displaying simulated I/O and your workstation keyboard is again the active simulated I/O input device. You can enter text followed by the < RETURN> , and the demo program will move the text to the simulated I/O display buffer.

### Closing the Keyboard Simulated I/O Demo

The simulated I/O demo program will deactivate the keyboard as an input device when the following key is pressed:

```
<RETURN>
```

Keyboard simulated I/O is now closed, and the rest of the simulated I/O demo program will begin executing. You will see the simulated I/O demo program test the "position cursor" command by selectively writing to a single display location. Then, the execute "UNIX command" is used to concatenate the **simiodemo.c** demo program to the simulated I/O "stdout" file (which was specified as "/dev/simio/display" in the simulated I/O configuration). Other simulated I/O commands are executed as well. Look at the **simiodemo.c** file (figure 4-3) to see exactly what the simulated I/O demo program does.



## Examples

### Running The Simulated I/O Demo Program

```
#include "simiodemo.h"

#define READ_BUF_SIZE 255
#define TRUE 1
#define FALSE 0

/* FORWARD DECLARATIONS */
extern int initsimio();
extern int open();
extern int close();
extern int read();
extern int write();
extern int clear_screen();
extern int pos_cursor();
extern int exec_cmd();
extern int kill();
extern int unlink();
extern long lseek();

unsigned char control_addr[300];
unsigned char *simio_addr;
int errno;

main()
{
    int fd1, fd2, fd3;          /* Declarations for the file descriptors. */
    int fd4;
    int pid1, pid2;           /* Declarations for process IDs. */
    int numb_read;
    unsigned char buf[READ_BUF_SIZE];

    simio_addr = control_addr;
    initsimio();

    /*
     *   Open stdout and stdin -- typically a display and
     *   keyboard. These can be set up during configuration
     *   to be the defaults of /dev/simio/display and
     *   /dev/simio/keyboard or changed to a file or
     *   directly to another terminal (/dev/tty00 for example).
     */

    fd1 = open("stdout", S_O_WRITE);
    fd4 = open("stdin", S_O_READ | S_O_NDELAY);

    write(fd1, "KEYBOARD NOW OPEN\n", 18);

    /*
     *   This loop reads the keyboard until a line containing
     *   only a <RETURN> is encountered.
     *
     *   Each line read is written to the display following
     *   the string "OUTPUT: ".
     */

    while (TRUE)
    {
```

Figure 4-3. The "simiodemo.c" Demo Program



Examples  
**Running The Simulated I/O Demo Program**

```
numb_read = read(fd4, buf, READ_BUF_SIZE);
if (numb_read > 0)
{
    /* keyboard input detected */
    write(fd1, "OUTPUT: ", 9);
    write(fd1, buf, numb_read);
    if (*buf == '\n')
    {
        break;
    }
}
}

close(fd4);
write(fd1, "KEYBOARD NOW CLOSED\n", 20);
close(fd1);

/* Examples of position cursor and clear screen commands. */

fd1 = open("/dev/simio/display", S_O_WRITE);
pos_cursor(fd1, 5, 30);
write(fd1, "TESTING #1 POSITION CURSOR COMMAND", 34);
pos_cursor(fd1, 5, 39);
write(fd1, "2", 1);
pos_cursor(fd1, 5, 39);
write(fd1, "3", 1);
pos_cursor(fd1, 5, 39);
write(fd1, "4", 1);
clear_screen(fd1);
close(fd1);
fd2 = open("stdout", S_O_WRITE | S_O_CREATE);

/*
 * Example of the CREATE and EXCL flags.
 *
 * If the file "simiodemo.out" does not already exist
 * the file is created and the message "File did not exist"
 * is written to the file "simiodemo.out". If the file
 * already existed it is removed then created and the
 * text "File already existed" is written to the file.
 * If the file cannot be opened, the message "Cannot open
 * simiodemo.out" is written to standard out.
 */

fd3 = open("simiodemo.out", S_O_RDWR | S_O_CREATE | S_O_EXCL);
if (fd3 < 0)
{
    if (errno == FILE_EXISTS)
    {
        unlink("simiodemo.out");
        fd3 = open("simiodemo.out", S_O_RDWR | S_O_CREATE | S_O_EXCL);
        write(fd3, "File already existed\n", 21);
    }
    else
    {
        write(fd2, "Cannot open simiodemo.out", 25);
    }
}
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

## Examples

### Running The Simulated I/O Demo Program

```
}
else
{
    write(fd3, "File did not exist\n", 19);
}

/*
 * Example of the position file command. -- This assumes that the
 * file "simiodemo.out" was successfully opened and written to
 * by the code above.
 */

/* Position from beginning of file. */
lseek(fd3, 0L, 0);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, 1L, 0);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, 2L, 0);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

/* Offset from the current position.
 * -- Note that this is the end of file
 * because of the reads which read to end of file.
 */

lseek(fd3, -1L, 1);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, -2L, 1);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, -3L, 1);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

/* Position from end of file. */
lseek(fd3, -1L, 2);
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

Examples  
**Running The Simulated I/O Demo Program**

```
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, -2L, 2);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

lseek(fd3, -3L, 2);
if ((numb_read = read(fd3, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);
}

/*
 * Example of using the execute HP-UX command.
 *
 * A pipe is connected to the standard output of the cat
 * command. This output is then written to the display.
 */

pid1 = exec_cmd("cat simiodemo.c", (int *) 0, &fd1, (int *) 0);

/*
 * The sleep command is executed as an example of the
 * kill command. A kill with signal 0 (NULL signal)
 * is issued to find the status of the sleep. The NULL
 * signal will not terminate the process but the return
 * value of the kill routine indicated if the process
 * exits or not.
 */

pid2 = exec_cmd("sleep 5", (int *) 0, (int *) 0, (int *) 0);
while ((numb_read = read(fd1, buf, READ_BUF_SIZE)) > 0)
{
    write(fd2, buf, numb_read);

    /* See if sleep is still alive. */

    if (kill(pid2, 0) == 0)
    {
        write(fd2, "\nProcess still alive\n", 21);
    }
    else
    {
        write(fd2, "\nProcess terminated\n", 20);
    }
}

write(fd2, "\nDEMO COMPLETE", 14);

close(fd1);
close(fd2);
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

## Examples

### Running The Simulated I/O Demo Program

```
        close(fd3);
    }
    int
    initsimio()
    /*
     * This command is not actually required, but it allows
     * the program to be stopped while simio files are still
     * open and restarted without side effects from the previously
     * opened files.
     *
     * RETURNS      0 for no error.
     *              -1 for errors during the simio reset command
     *              and the global errno is set to the error code.
     */
    {
        *simio_addr = S_RESET;

        while (*simio_addr == S_RESET)
        {
            /* Empty loop - wait for results */
        }

        if (*simio_addr == 0)
        {
            return 0;
        }
        else
        {
            errno = (unsigned int) *simio_addr;
            return -1;
        }
    }
    int
    open(path, open_option)
    char *path;
    int open_option;
    /*
     * RETURNS      file descriptor >= 0 -- if the open succeeded.
     *              -1 if the open failed, errno is set the the
     *              error code.
     */
    {
        unsigned char *addr_ptr;

        addr_ptr = simio_addr + 1;

        *addr_ptr = open_option;
        addr_ptr++;

        *addr_ptr = 255;
        addr_ptr++;

        while (*path != '\0')
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

```
{
    *addr_ptr = *path;
    addr_ptr++;
    path++;
}

*addr_ptr = '\\0';

*simio_addr = S_OPEN;

while (*simio_addr == S_OPEN)
{
    /* Empty loop - wait for results */
}

if (*simio_addr == 0)
{
    return (unsigned int) *(simio_addr + 1);
}
else
{
    errno = (unsigned int) *simio_addr;
    return -1;
}
}

int
close(file_des)
int file_des;

/*
 * RETURNS      0 for no error.
 *              -1 for error and the global errno is set to the error code.
 */

{
    *(simio_addr + 1) = file_des;
    *simio_addr = S_CLOSE;

    while (*simio_addr == S_CLOSE)
    {
        /* Empty loop - wait for results */
    }

    if (*simio_addr == 0)
    {
        return 0;
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}

int
read(fd, buffer, nbytes)
int fd;
unsigned char *buffer;
```

Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)

## Examples

### Running The Simulated I/O Demo Program

```
int nbytes;

/*
 * RETURNS      number of bytes read (>= 0) if sucessfull.
 *             -1 for error and the global errno is set to the error code.
 */

{
    unsigned char *addr_ptr;
    unsigned char index;

    addr_ptr = simio_addr + 1;
    *addr_ptr = fd;
    addr_ptr++;
    *addr_ptr = nbytes;
    addr_ptr++;

    *simio_addr = S_READ;

    while (*simio_addr == S_READ)
    {
        /* Empty loop - wait for results */
    }

    addr_ptr = simio_addr + 3;

    for (index = 0; index < (*(simio_addr + 2)); index++)
    {
        *buffer = *addr_ptr;
        addr_ptr++;
        buffer++;
    }

    if (*simio_addr == 0)
    {
        return (unsigned int) *(simio_addr + 2);
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}

int
write(fd, buffer, nbytes)
int fd;
unsigned char *buffer;
int nbytes;

/*
 * RETURNS      number of bytes written (>= 0) if sucessfull.
 *             -1 for error and the global errno is set to the error code.
 */

{
    unsigned char *addr_ptr;
    int index;
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

Examples  
**Running The Simulated I/O Demo Program**

```
addr_ptr = simio_addr + 1;
*addr_ptr = fd;
addr_ptr++;
*addr_ptr = nbytes;
addr_ptr++;

for (index = 0; index < nbytes; index++)
{
    *addr_ptr = *buffer;
    addr_ptr++;
    buffer++;
}

*simio_addr = S_WRITE;

while (*simio_addr == S_WRITE)
{
    /* Empty loop - wait for results */
}

if (*simio_addr == 0)
{
    return (unsigned int) *(simio_addr + 2);
}
else
{
    errno = (unsigned int) *simio_addr;
    return -1;
}
}

int
clear_screen(fd)
int fd;

/*
 * RETURNS      0 if successfull.
 *              -1 for error and the global errno is set to the error code.
 */
{
    *(simio_addr + 1) = fd;
    *simio_addr = S_CLEAR_DISP;

    while (*simio_addr == S_CLEAR_DISP)
    {
        /* Empty loop - wait for results */
    }

    if (*simio_addr == 0)
    {
        return 0;
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}
}
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

## Examples

### Running The Simulated I/O Demo Program

```
int
pos_cursor(fd, line, col)
int fd;
int line;
int col;

/*
 * RETURNS      0 if successfull.
 *              -1 for error and the global errno is set to the error code.
 */

{
    unsigned char *addr_ptr;

    addr_ptr = simio_addr + 1;
    *addr_ptr = fd;
    addr_ptr++;
    *addr_ptr = line;
    addr_ptr++;
    *addr_ptr = col;

    *simio_addr = S_POS_CURSOR;

    while (*simio_addr == S_POS_CURSOR)
    {
        /* Empty loop - wait for results */
    }

    if (*simio_addr == 0)
    {
        return 0;
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}

int
exec_cmd(command, file1, file2, file3)
char *command;
int *file1, *file2, *file3;

/*
 * PARAMETERS   command - a string containing the name and parameters
 *              of the command to execute.
 *              file1, file2, file3 - pointers to variables to return
 *              the file descriptors of the pipes connected to the
 *              stdin, stdout, and stderr of the process executed.
 *              If any pointer is NULL, that pipe is connected to /dev/null
 *              and no file descriptor is returned.
 *
 * RETURNS      process id (>=0) if successfull and the file descriptors of
 *              the pipes are returned in file1, file2, and file3.
 *              -1 for error and the global errno is set to the error code.
 */
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**



```
{
    unsigned char *addr_ptr;
    unsigned char bitmap = 0;

    if (file1 != 0)
    {
        bitmap |= 1;
    }
    if (file2 != 0)
    {
        bitmap |= 2;
    }
    if (file3 != 0)
    {
        bitmap |= 4;
    }

    addr_ptr = simio_addr + 1;
    *addr_ptr = bitmap;
    addr_ptr++;
    *addr_ptr = 255;
    addr_ptr++;

    while (*command != '\0')
    {
        *addr_ptr = *command;
        command++;
        addr_ptr++;
    }

    *addr_ptr = '\0';

    *simio_addr = S_EXEC_CMD;

    while (*simio_addr == S_EXEC_CMD)
    {
        /* Empty loop - wait for results */
    }

    if (file1 != 0)
    {
        *file1 = *(simio_addr + 2);
    }
    if (file2 != 0)
    {
        *file2 = *(simio_addr + 3);
    }
    if (file3 != 0)
    {
        *file3 = *(simio_addr + 4);
    }

    if (*simio_addr == 0)
    {
        return (unsigned int) *(simio_addr + 1);
    }
    else
    {

```

Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)



## Examples

### Running The Simulated I/O Demo Program

```
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}
int
kill(pid, sig)
int pid, sig;

/*
 * RETURNS      0 if the process exists.
 *              -1 for error and the global errno is set to the error code.
 */

{
    *(simio_addr + 1) = pid;
    *(simio_addr + 2) = sig;
    *simio_addr = S_KILL;

    while (*simio_addr == S_KILL)
    {
        /* Empty loop - wait for results */
    }

    if (*simio_addr == 0)
    {
        return 0;
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}

int
unlink(path)
char *path;

/*
 * RETURNS      0 if successfull.
 *              -1 for error and the global errno is set to the error code.
 */

{
    unsigned char *addr_ptr;

    addr_ptr = simio_addr + 1;
    *addr_ptr = 255;
    addr_ptr++;

    while (*path != '\0')
    {
        *addr_ptr = *path;
        addr_ptr++;
        path++;
    }

    *addr_ptr = '\0';
}
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

Examples  
**Running The Simulated I/O Demo Program**

```
*simio_addr = S_DELETE_FILE;

while (*simio_addr == S_DELETE_FILE)
{
    /* Empty loop - wait for results */
}

if (*simio_addr == 0)
{
    return 0;
}
else
{
    errno = (unsigned int) *simio_addr;
    return -1;
}
}

long
lseek(fd, offset, whence)
int fd;
long offset;
int whence;

/*
 * RETURNS      non-negative integer indicating file pointer if successfull.
 *              -1 for error and the global errno is set to the error code.
 */

{
    unsigned char *addr_ptr;

    addr_ptr = simio_addr + 1;
    *addr_ptr = fd;
    addr_ptr++;

    *addr_ptr = offset & 0xff;
    addr_ptr++;
    *addr_ptr = (offset >> 8) & 0xff;
    addr_ptr++;

    /* The following code checks to see if the compiler for this
     * processor supports 32 bit (4 byte) longs.  If so, the offset
     * parameter is used.  Otherwise only 16 bits are available and
     * the offset is sign extended to created a 32 bit integer.  This
     * code must be altered if files greater than 64K bytes long must
     * be supported with compilers which do not support 32 bit integers.
     */

    if (sizeof(long) >= 4)
    {
        *addr_ptr = (offset >> 16) & 0xff;
        addr_ptr++;
        *addr_ptr = (offset >> 24) & 0xff;
        addr_ptr++;
    }
    else if (offset >= 0)
    {
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

## Examples

### Running The Simulated I/O Demo Program

```
        /* Compiler does not support 32 bit longs - sign extend.      */
        *addr_ptr = 0;
        addr_ptr++;
        *addr_ptr = 0;
        addr_ptr++;
    }
    else
    {
        *addr_ptr = 0xff;
        addr_ptr++;
        *addr_ptr = 0xff;
        addr_ptr++;
    }
    *addr_ptr = whence;
    *simio_addr = S_POS_FILE;
    while (*simio_addr == S_POS_FILE)
    {
        /* Empty loop - wait for results */
    }
    if (*simio_addr == 0)
    {
        if (sizeof(long) >= 4)
        {
            /* Compiler supports 32 bit longs -- the offset
             * is already guaranteed to be non-negative.
             */
            return ((*(simio_addr + 2) & 0xff)
                | (*(simio_addr + 3) << 8) & 0xff)
                | (*(simio_addr + 4) << 16) & 0xff)
                | (*(simio_addr + 5) << 24) & 0xff);
        }
        else
        {
            /* Truncate most significant bits of absolute offset
             * to 15 bits to guarantee that number is non-negative.
             */
            return (*(simio_addr + 2) & 0xff) | ((*(simio_addr + 3) << 8) &
0x7f);
        }
    }
    else
    {
        errno = (unsigned int) *simio_addr;
        return -1;
    }
}
```

**Figure 4-3. The "simiodemo.c" Demo Program (Cont'd)**

## RS-232 Simulated I/O

Simulated I/O is a technique for reading and writing to files or devices from an emulator. A segment of emulation RAM is dedicated to the simulated I/O and shared by the emulator and the emulation system controller. To cause simulated I/O to occur, the emulator writes a command byte into the initial location of the shared segment, called the Control Address (CA). The emulation software polls this location continuously searching for commands from the emulator. When a command is recognized, the action is carried out by the emulation software on the host computer. Simulated I/O actions include open, close, read and write to a file. Since devices are treated as files in UNIX, simulated I/O to a device is similar to simulated I/O to a file.

## Configuring RS-232 Lines for Simulated I/O

In order to use an RS-232 line on the UNIX host computer for simulated I/O from an emulator the line must first be configured, it can then be treated as a file. An understanding of how serial lines are dealt with in UNIX systems is an important prerequisite to configuration.

### Serial Lines in UNIX Systems

When a UNIX system is booted, the initial process is called "init". This process always has process ID 1 and is the parent or ancestor of every other process running on the system. In the subdirectory /etc a file named inittab lists all the devices to be initialized by init when the system starts up. This file will contain entries for each serial line specifying the initial state of the line, whether to spawn a process for line and what to do if that process terminates. If a line is listed as requiring a process, init spawns a process called getty for that line. Getty sets the communication parameters for that line and prints the banner and login prompt on the line. When a login name is typed in, getty spawns a login process which asks for the password. If the password is correct, login forks a shell (command interpreter) process and attaches the standard input and standard output file descriptors for that shell to the /dev entries for that particular serial line.



### Serial Lines Used with Simulated I/O

Using a serial line device file as a read/write file from an emulator may require a different sequence of events in the startup of that line. UNIX avoids deadlock problems by permitting race conditions to occur. Two processes can simultaneously read and write the same file, assuming both have read/write permission, without complaint from the operating system. This does not imply that there will be no problems. Two processes reading the same device file will get alternate characters. If a shell is running on a given serial line and a user attempts simulated I/O reads from that line, both the shell and the emulator will get part of the input. If both the shell and an emulator write to a serial line, the output will be interleaved. There are several ways of solving this problem, depending on the operating environment.

### Serial Lines for User Terminals and Simulated I/O

The first is the situation where it is desired to generally use the serial line as a user terminal line and occasionally use it for simulated I/O. In this case the `/etc/inittab` file would probably contain a line to initialize the serial line for the `getty`, `login`, `shell` sequence of processes. The line can then be used for standard login sessions. If the user wishes to make the line available to the emulator for simulated I/O reads, the he can simply issue a `sleep(1)` command which will suspend the shell process for whatever period of time the user wishes. While the shell is suspended the emulator can read the serial line without interference from the shell, since the shell will issue no reads while it is suspended. Anything typed at the terminal could then be read by the emulator. If the user wished to use the line to communicate with some other device, for example, some other computer, an RS-232 ABC switch could be used to switch either the terminal or the computer to the serial line.

There is one potential problem with switching lines in this manner. The serial lines on the UNIX host computer can be configured to send a hang-up signal to the attached process if the DTR line is dropped. This is in fact the normal configuration. Switching the line from one device to another will cause DTR to be momentarily disconnected which will be interpreted as a dropped line. This will cause a hang-up signal to be sent to the shell, which will murder its siblings (`sleep`) and exit. When the shell exits, `init` will respawn a

getty process for that line which will issue a read on the serial line, creating a race condition between getty and the emulator program. This problem can be eliminated by reconfiguring the serial line, which requires powering the machine down, removing the serial line card and changing the DIP switch configuration. Alternately a special cable can be wired which will loop the modem control signals back to the serial line, making it impossible for the line to be dropped. Simply wire pins 4, 5, and 6 together and 8 to 20 on the connector to the UNIX host computer.

### Dedicated Serial Lines

Another possibility is that a line is to be dedicated to communication with some external device and it is desired that the getty, login, shell sequence never occur on that line. The line is then configured to be "off" in the /etc/inittab file. In this configuration the line is initialized to standard defaults (300 baud, no parity, etc.). These defaults are nearly useless and were chosen when UNIX was an infant and have been kept around for historic reasons, there being no good reason for their existence. Another unfortunate fact is that there is no way to modify these defaults since they are hard-wired into the UNIX kernel. The most direct way to change the configuration of a port is to use the stty(1) command. The action of this command can be directed to any terminal so it can change the defaults of any terminal. Unfortunately, as soon as the stty command terminates it will close the device file at which time it will return to its default (useless) configuration. A workaround for this problem is to issue a stty command that will not terminate until the emulation process opens the device file, for example:

```
(stty -modes; sleep 100000) /dev/tty?? &
```

In this sample command, -modes refers to the list (possibly long list) of modes the user wishes to apply to the line. The argument to sleep is the number of seconds to suspend the process. By grouping the stty and sleep commands, the affected device file will not be closed until both commands terminate. There are some drawbacks to this method. The command itself is not intuitive and will probably have to be killed manually at some later point. There is also no way to prevent another user from reading or writing the device file or changing its configuration with stty.

Another way to deal with this problem is to write a program that will configure the serial line. Since simulated I/O can spawn new processes, the program can be started and halted from within the program using simulated I/O. Using a program to configure the device file also presents an opportunity to create lock files to prevent race conditions between two processes trying to communicate with the same device. The `cu(1)` utility uses this technique to prevent multiple access to a single serial line.

### Example Programs

Two example programs for configuring ports are presented here. The first, "useport.c" (figure 4-4), creates a lock file and configures the port. It then copies its standard input to the device file and writes any characters appearing as input to the device file to its standard output. Useport.c acts as a filter program. Communicating with a device file becomes a simple matter of starting up the useport process and communicating with its standard input and standard output.

The second example program, "setport.c" (figure 4-5), is nearly identical to useport.c. The only difference is that setport.c never reads or writes the device file, it merely configures the file and then sleeps. After the setport process is started, the device file can be opened directly for reads and writes. This method has the convenience of using a process to configure a port without the overhead of passing all the characters through an additional filter process.

### Special Considerations

There are some special considerations when using simulated I/O to communicate with processes. If a process started by simulated I/O exits, there is no explicit notification of that fact. The problem is most severe when the process is opened for no wait reads. In this case a read which returns a length of zero does not indicate end of file, which is the normal way of determining that a process has terminated. When the process is started, its standard error output can be connected to a simulated I/O file descriptor. The process can indicate that it is exiting by writing a message to this file. It is also possible to use the simulated I/O "kill process" command to determine if the process is still alive. Simply send signal 0, which has no effect, a return value of 0 indicates that the process is still alive.



It is always a good idea to execute the simulated I/O "reset" command as the first simulated I/O command in a program. Simulated I/O reset will clean up any remains of a possible previous execution. This is particularly true when using simulated I/O to communicate with processes. The simulated I/O Reset command will send signal SIGPIPE to each simulated I/O process. It is important to catch this signal, do any necessary clean up and exit in any processes spawned by simulated I/O.

One final consideration is necessary if simulated I/O is to be used for machine to machine communication. Not all RS-232 devices for UNIX host computers are sufficiently buffered for reliable machine to machine communication. Single character buffered devices cannot be reliably used for machine to machine communication.



## Examples

### RS-232 Simulated I/O

```
/* USEPORT.C
   Connect the standard input to a tty output
   and the standard output to a tty input.
   Create a lock file to prevent simultaneous
   access to the tty.
*/
#include <stdio.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <termio.h>
#include <fcntl.h>
#define MIN 1      /* minimum chars to buffer for reads */
#define TIME 1    /* delay for reads */

int fd, arg;
struct termio old;
char lock[256];

sig_func()
{
    /* restore stdin file mode */
    fcntl(0, F_SETFL, arg);
    /* restore terminal settings */
    ioctl(fd, TCSETA, &old);
    /* remove the lock file */
    unlink(lock);
    exit(1);
}

main(argc, argv)
int argc;
char *argv[];
{
    int i, count;
    FILE *fp;
    char c[1], buf[512], *p;
    struct termio new;

    if(argc != 2){
        fprintf(stderr, "Usage: %s ttyname\\n", argv[0]);
        exit(1);
    }
    if((fd = open(argv[1], O_RDWR | O_NDELAY)) == -1){
        fprintf(stderr, "Attempt to open %s failed\\n", argv[1]);
        exit(1);
    }
    if(isatty(fd) == 0){
        fprintf(stderr, "%s is not a tty\\n", argv[1]);
        exit(1);
    }

    /* Create a lock file */
    strcpy(lock, "/usr/spool/uucp/LCK..");
    p = ttyname(fd);

    /* this is not a nice way to skip over "/dev/" */
    p += 5;
    strcat(lock, p);
}
```

Figure 4-4. The "useport.c" RS-232 Sim. I/O Example

```

if((i = open(lock, O_RDONLY | O_CREAT | O_EXCL, 0444)) == -1){
    fprintf(stderr,"Cannot create lock file %s\n", lock);
    exit(1);
}
close(i);

/* get the tty status */
ioctl(fd, TCGETA, &old);
new = old;

/* set up new terminal line characteristics */
new.c_cflag = B9600 | /* 9600 baud */
              CS7 | /* 7 data bits */
              /* stop bits defaults to 1 */
              CREAD | /* enable receiver */
              PARENB | /* enable parity */
              /* parity defaults to even */
              CLOCAL; /* local line */
new.c_lflag = 0;
new.c_oflag = 0;
new.c_iflag = IGNBRK | /* ignore break condition */
              IGNPAR | /* ignore parity errors */
              ISTRIP | /* strip all chars to 7 bits */
              IXON | /* enable start/stop output control */
              IXOFF; /* enable start/stop input control */
new.c_cc[4] = MIN;
new.c_cc[5] = TIME;
ioctl(fd, TCSETA, &new);

/* set no delay on stdin */
arg = fcntl(0, F_GETFL, arg);
fcntl(0, F_SETFL, O_NDELAY | arg);

/* catch all signals */
for(i = 1; i < 24; i++)
    signal(i, sig_func);

while(1){
    if((count = read(0, buf, 1)) == -1){
        fprintf(stderr,"Error reading stdin\n");
        sig_func();
        exit(1);
    }
    if(count){
        write(fd, buf, count);
    }
    if((count = read(fd, buf, 1)) == -1){
        fprintf(stderr,"Error reading %s\n", argv[1]);
        sig_func();
        exit(1);
    }
    if(count)
        write(1, buf, count);
}
}

```

Figure 4-4. The "useport.c" Example (Cont'd)

## Examples

### RS-232 Simulated I/O

```
/* SETPORT.C
   Set terminal characteristics.
   Create a lock file to prevent simultaneous
   access to the tty.
*/
#include <stdio.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <termio.h>
#include <fcntl.h>
#define MIN 1 /* minimum chars to buffer for reads */
#define TIME 1 /* delay for reads */

int fd, arg;
struct termio old;
char lock[256];

sig_func()
{
    /* restore terminal settings */
    ioctl(fd, TCSETA, &old);
    /* remove the lock file */
    unlink(lock);
    exit(1);
}

main(argc, argv)
int argc;
char *argv[];
{
    int i, count;
    FILE *fp;
    char c[1], buf[512], *p;
    struct termio new;

    if(argc != 2){
        fprintf(stderr, "Usage: %s ttyname\\n", argv[0]);
        exit(1);
    }
    if((fd = open(argv[1], O_RDWR | O_NDELAY)) == -1){
        fprintf(stderr, "Attempt to open %s failed\\n", argv[1]);
        exit(1);
    }
    if(isatty(fd) == 0){
        fprintf(stderr, "%s is not a tty\\n", argv[1]);
        exit(1);
    }

    /* Create a lock file */
    strcpy(lock, "/usr/spool/uucp/LCK..");
    p = ttyname(fd);

    /* this is not a nice way to skip over "/dev/" */
    p += 5;
    strcat(lock, p);
    if((i = open(lock, O_RDONLY | O_CREAT | O_EXCL, 0444)) == -1){
        fprintf(stderr, "Cannot create lock file %s\\n", lock);
        exit(1);
    }
}
```

Figure 4-5. The "setport.c" RS-232 Sim. I/O Example

```
}
close(i);

/* get the tty status */
ioctl(fd, TCGETA, &old);
new = old;

/* set up new terminal line characteristics */
new.c_cflag = B9600 | /* 9600 baud */
              CS7 | /* 7 data bits */
              /* stop bits defaults to 1 */
              CREAD | /* enable receiver */
              PARENB | /* enable parity */
              /* parity defaults to even */
              CLOCAL; /* local line */
new.c_lflag = 0;
new.c_oflag = 0;
new.c_iflag = IGNBRK | /* ignore break condition */
              IGNPAR | /* ignore parity errors */
              ISTRIP | /* strip all chars to 7 bits */
              IXON | /* enable start/stop output control */
              IXOFF; /* enable start/stop input control */
new.c_cc[4] = MIN;
new.c_cc[5] = TIME;
ioctl(fd, TCSETA, &new);

/* catch all signals */
for(i = 1; i < 24; i++)
    signal(i, sig_func);

while(1){
    /* nap */
    sleep(1000);
}
}
```

Figure 4-5. The "setport.c" Example (Cont'd)





## Error Codes

---

### Overview

This chapter contains:

- A table of the simulated I/O error codes.
- A description of the errors as they relate to the simulated I/O commands.

---

### Introduction

This chapter contains a list of all simulated I/O error codes and a description of when they can occur and what condition causes each error.



Table 5-1. Simulated I/O Error Codes

ERROR CODE	ERROR NAME
03H	FILE NOT FOUND
04H	FILE ALREADY EXISTS
08H	CANNOT READ MEMORY
09H	INVALID COMMAND
0BH	INVALID ROW OR COLUMN
0FH	INVALID FILE NAME
11H	NO FREE DESCRIPTORS
12H	INVALID FILE DESCRIPTOR
16H	NO PERMISSION
17H	INVALID OPTIONS
18H	TOO MANY FILES
19H	NO FREE PROCESS ID
1AH	TOO MANY PROCESSES
1BH	INVALID COMMAND NAME
1CH	INVALID PROCESS ID
1DH	INVALID SIGNAL
1EH	NO SUCH PROCESS
1FH	NO SEEK ON PIPE
7EH	UNIX ERROR
7FH	CONTIUE ERROR



**Error Code  
Description (By  
Simulated I/O  
Command)**

**General Errors**

**08H CANNOT READ MEMORY.** The read of the command code from memory failed.

**09H INVALID COMMAND.** Command code is not in the range 90H through 9BH.

**7EH UNIX ERROR.** This error means that some UNIX system call has failed. When a system call fails, the reason for the failure is indicated in a global variable `errno` — see `errno (2)` in the UNIX reference manual. When simulated I/O returns UNIX ERROR, the value of `errno` is returned in location `CA + 2` (`CA + 1`, which contains the file descriptor, is never modified). No other error codes return `errno`.

**7FH CONTINUE ERROR.** If emulation is exited and then re-entered for a continued session, all simulated I/O files will be closed and all simulated I/O processes will no longer be executing. Any simulated I/O command that requires an open file (read, write, position cursor, etc.) issued by your program after emulation has been re-entered will get the CONTINUE ERROR error code. (Commands like `open`, and UNIX command will proceed with no errors.) The simulated I/O reset command will prevent additional CONTINUE ERRORS from occurring in the continued emulation session.

**Open (90H)  
Simulated I/O Errors**

**0FH INVALID FILE NAME.** File name length = 0.

**11H NO FREE DESCRIPTORS.** Too many simulated I/O files are open. The maximum is 12.



**Error Code Description (By Simulated I/O Command)**

**Open File Errors**

**03H FILE NOT FOUND.** UNIX open (2) failed, and errno = ENOENT.

**04H FILE ALREADY EXISTS.** UNIX open (2) failed, and errno = EEXIST (create and exclusive options given).

**16H NO PERMISSION.** UNIX open (2) failed, and errno = EACCES.

**17H INVALID OPTIONS.** The open options do not contain exactly one of: read only (0), write only (1), or read-write (2). Open options not valid.

**18H TOO MANY FILES.** UNIX open (2) failed, and errno = EMFILE (NFILE UNIX descriptors are open by emulation).

**7EH UNIX ERROR.** Any error from close (2).

**Close (91H)  
Simulated I/O Errors**

**12H INVALID FILE DESCRIPTOR.** File descriptor indicated not open.

**7FH CONTINUE ERROR.** Attempt to close any file descriptor after continue. (See description under "General Errors".)

**Close File Errors**

**7EH UNIX ERROR.** Any error from close (2).

**Read (92H)  
Simulated I/O Errors**

**09H INVALID COMMAND.** Attempt to read from the display.

**12H INVALID FILE DESCRIPTOR.** File descriptor indicated not open.

**7FH CONTINUE ERROR.** Attempt to read anything after continuation. (See description under "General Errors".)

**Read File Errors**

**7EH UNIX ERROR.** Any error from read (2).

**Error Code Description (By Simulated I/O Command)****Write (93H)  
Simulated I/O Errors****08H CANNOT READ MEMORY.** Cannot read buffer to write.**09H INVALID COMMAND.** Attempt to write to keyboard.**12H INVALID FILE DESCRIPTOR.** File descriptor indicated not open.**7FH CONTINUE ERROR.** Attempt to write anything after continuation. (See description under "General Errors".)**Read File Errors****7EH UNIX ERROR.** Any error from write(2).**Delete File (94H)  
Simulated I/O Errors****03H FILE NOT FOUND.** UNIX unlink (2) failed and errno set to ENOENT.**08H CANNOT READ MEMORY.** Read of file name failed.**0FH INVALID FILE NAME.** File name length = 0, or unlink (2) failed (errno set to ENOTDIR).**16H NO PERMISSION.** UNIX unlink (2) failed and errno set to EACCES.**7EH UNIX ERROR.** UNIX unlink (2) failed for some other reason.**Position File (95H)  
Simulated I/O Errors****09H INVALID COMMAND.** File descriptor indicates open file of type other than a UNIX file.**12H INVALID FILE DESCRIPTOR.** File descriptor indicated not open.**17H INVALID OPTIONS.** Start code is not 0, 1, or 2, or lseek (2) failed and errno set to EINVAL.**1FH NO SEEK ON PIPE.** File descriptor indicates a pipe to a UNIX command.

Error Codes

**Error Code Description (By Simulated I/O Command)**

**7FH CONTINUE ERROR.** A position file attempted after continuation. (See description under "General Errors".)

**Position Cursor (96H)  
Simulated I/O Errors**

**09H INVALID COMMAND.** Attempt to position cursor on file that is not a display.

**0BH INVALID ROW OR COLUMN.** The row number is greater than or equal to 50 rows, or the column number is greater than or equal to 80 columns (or the number of columns on the display whichever is greater).

**12H INVALID FILE DESCRIPTOR.** Attempt to position cursor on file that is not open.

**7FH CONTINUE ERROR.** Attempt to position cursor after continuation. (See description under "General Errors".)

**Clear Display (97H)  
Simulated I/O Errors**

**09H INVALID COMMAND.** Attempt to clear display on file that is not a display.

**12H INVALID FILE DESCRIPTOR.** Attempt to clear display on file that is not open.

**7FH CONTINUE ERROR.** Attempt to clear display after continuation. (See description under "General Errors".)

**UNIX Command  
(98H) Simulated I/O  
Errors**

**08H CANNOT READ MEMORY.** Read failed on read of command name.

**11H NO FREE DESCRIPTORS.** Simulated I/O descriptor table is full.

**18H TOO MANY FILES.** UNIX pipe (2) failed.

**19H NO FREE PROCESS ID.** The maximum number of processes are already active.

**1AH TOO MANY PROCESSES.** UNIX fork (2) failed and errno = EAGAIN.

**Error Code Description (By Simulated I/O Command)**

**1BH INVALID COMMAND NAME.** Command name length is 0.

**7EH UNIX ERROR.** UNIX fork (2) failed and errno does not equal EAGAIN.

**Kill Process (99H)  
Simulated I/O Errors**

**16H NO PERMISSION.** Kill failed and errno = EPERM.

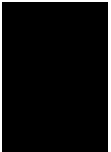
**1CH INVALID PROCESS ID.** The simulated I/O process id is unused or out of range (the simulated I/O process entry does not exist).

**1DH INVALID SIGNAL.** Kill failed and errno = EINVAL.

**1EH NO SUCH PROCESS.** Kill failed and errno = ESRCH (the UNIX process does not exist).

**7EH UNIX ERROR.** Kill failed with other error number.





# Index

---

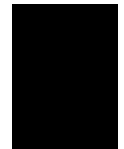
- A**
  - absolute byte offset (position file command), **30**
  - absolute file for demo program, loading the, **52**
  - activating the simulated I/O display, **52**
  - activating the simulated I/O keyboard, **53**
  - active processes, maximum number of, **17**
  - append (open file option), **22**
  - append option and positioning in files, **31**
- B**
  - buffer (Control Address), length of, **10**
  - buffer (Control Address), locating in memory, **10**
  - bytes to read, maximum number, **26**
- C**
  - column number (position cursor command), **33**
  - columns in display simulated I/O buffer, **43**
  - command name (execute UNIX command), **36**
  - command name length (execute UNIX command), **36**
  - compiling the simulated I/O demo program, **50**
  - configuring simulated I/O for the demo program, **51**
  - control address (CA) buffer, length of, **10**
  - control address (CA) buffer, locating in memory, **10**
  - control addresses, maximum number of, **17**
  - copying the simulated I/O demo program, **50**
  - create (open file option), **22**
- D**
  - delay, no (open file option), **22**
  - demo program, configuring simulated I/O, **51**
  - demo simulated I/O program, copying the, **50**
  - descriptor, file, **23**
  - dev/simio/display, default for standard error output, **15**
  - dev/simio/display, default for standard output, **15**
  - dev/simio/keyboard, default for standard input, **15**
  - display simulated I/O, clearing the display, **45**
  - display simulated I/O, closing, **46**
  - display simulated I/O, number of lines and columns, **43**
  - display simulated I/O, opening, **42**
  - display simulated I/O, positioning the cursor, **44**

## Index

- display simulated I/O, writing, **43**
- display simulated\_io emulation command, **42**
- E** emulation ram, loading the control address into, **15**  
entering the measurement system for the demo program, **51**  
exclusive (open file option), **22**
- F** file descriptor, **23**  
file descriptor for stderr (execute UNIX command), **37**  
file descriptor for stdin (execute UNIX command), **36**  
file descriptor for stdout (execute UNIX command), **37**
- I** ID, simulated I/O process, **36**
- K** keyboard simulated I/O, closing, **49**  
keyboard simulated I/O, opening, **46**  
keyboard simulated I/O, reading, **48**  
keyboard simulated I/O, reading from, **25**  
keyboard simulated I/O, terminating a line of input, **47**
- L** length of path name (open command), **23**  
length of the control address buffer, **10**  
line number (position cursor command), **33**  
lines in display simulated I/O buffer, **43**  
loading the demo program absolute file, **52**  
locating control address into emulation ram, **15**  
locating the control address buffer in memory, **10**
- M** maximum number of bytes that can be read, **26**  
maximum number of bytes that can be written, **27**  
maximum number of control addresses, **17**  
maximum number of open files, **17**  
measurement system for demo program, entering the, **51**  
modify keyboard\_to\_simio emulation command, **47**
- N** no delay (open file option), **22**
- O** open file option, **21**  
open files, maximum number of, **17**  
open pipe specification (execute UNIX command), **36**  
opening display simulated I/O, **42**
- P** path name (open command), **23**  
pipe, open specification (execute UNIX command), **36**  
positioning in files opened with the append option, **31**



- process ID (execute UNIX command), **36**
- processes (active), maximum number of, **17**
- protocol (general), **10**
  
- R**
  - read command, maximum number of bytes that can be read, **26**
  - read only (open file option), **22**
  - reading from keyboard simulated I/O, **25**
  - real-time, **9**
  - relative byte offset (position in file command), **31**
  - reserved file names (stdin, stdout, and stderr), **15**
  - reserved files, attempting to delete, **29**
  - RS-232 simulated I/O, **69/77**
  
- S**
  - signal to send to process (kill command), **38**
  - SIGPIPE, **39**
  - simio\_ca\_xxx, symbols associated with control addresses, **15**
  - simulated I/O commands, list of, **19**
  - simulated i/o control address? (configuration questions), **14**
  - simulated I/O demo program, copying the, **50**
  - simulated I/O demo, configuring, **51**
  - simulated I/O process ID (execute UNIX command), **36**
  - softkeys (when keyboard simulated I/O is active), **48**
  - special files, attempting to delete, **29**
  - starting location (position in file command), **31**
  - stderr file descriptor (execute UNIX command), **37**
  - stderr, reserved file name, **15**
  - stdin file descriptor (execute UNIX command), **36**
  - stdin, reserved file name, **15**
  - stdout file descriptor (execute UNIX command), **37**
  - stdout, reserved file name, **15**
  - suspend softkey, **48**
  
- T**
  - terminating an input line (keyboard simulated I/O), **47**
  - truncate (open file option), **22**
  
- U**
  - using display simulated I/O, **42**
  
- W**
  - write command, maximum number of bytes that can be written, **27**
  - write only (open file option), **22**





---

## Certification and Warranty

**Certification** Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

**Warranty** This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

**Limitation of Warranty** The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

**No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.**

**Exclusive Remedies**

**The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.**

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.