TEK REFERENCE MANUAL

Part No. 070-5604-00
Product Group 18

4404
ARTIFICIAL INTELLIGENCE SYSTEM

Please Check for CHANGE INFORMATION at the Rear of this Manual

First Printing DEC 1984

Tektronix®
COMMITTED TO EXCELLENCE

WARRANTY FOR SOFTWARE PRODUCTS

Tektronix warrants that this software product will conform to the specifications set forth herein, when used properly in the specified operating environment, for a period of three (3) months from the date of shipment, or if the program is installed by Tektronix, for a period of three (3) months from the date of installation. If this software product does not conform as warranted, Tektronix will provide the remedial services specified below. Tektronix does not warrant that the functions contained in this software product will meet Customer's requirements or that operation of this software product will be uninterrupted or error-free or that all errors will be corrected.

In order to obtain service under this warranty, Customer must notify Tektronix of the defect before the expiration of the warranty period and make suitable arrangements for such service in accordance with the instructions received from Tektronix. If Tektronix is unable, within a reasonable time after receipt of such notice, to provide the remedial services specified below, Customer may terminate the license for the software product and return this software product and any associated materials to Tektronix for credit or refund.

This warranty shall not apply to any software product that has been modified or altered by Customer. Tektronix shall not be obligated to furnish service under this warranty with respect to any software product a) that is used in an operating environment other than that specified or in a manner inconsistent with the Users Manual and documentation or b) when the software product has been integrated with other software if the result of such integration increases the time or difficulty of analyzing or servicing the software product or the problems ascribed to the software product.

TEKTRONIX DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. TEKTRONIX RESPONSIBILITY TO PROVIDE REMEDIAL SERVICE WHEN SPECIFIED, REPLACE DEFECTIVE MEDIA OR REFUND CUSTOMER'S PAYMENT IS THE SOLE AND EXCLUSIVE REMEDY PROVIDED TO CUSTOMER FOR BREACH OF THIS WARRANTY. TEKTRONIX WILL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES IRRESPECTIVE OF WHETHER TEKTRONIX HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

MANUAL REVISION STATUS

PRODUCT: 4404 Artificial Intelligence System

This manual supports the following versions of this product: Serial Numbers B010100 and up.

REV DATE	DESCRIPTION
DEC 1984	Original Issue

CONTENTS

Section 5 THE ASSEMBLER AND LINKING LOADER

Section 6 SYSTEM CALLS

Section 8 4404 HARDWARE SUPPORT

Section 10 TERMINAL EMULATION

INDEX

TABLES

Section 1

INTRODUCTION

ABOUT THIS MANUAL

This manual is the primary user's and programmer's reference to the 4404 operating system and hardware support. This manual contains concise summaries of the commands and utilities included with your 4404 as standard software, and a summary of how to invoke and use each command. This manual does not attempt to show you how to put commands together to perform a task; that information is covered in the 4404 User's Manual. The User's Manual also contains a complete list of the other manuals available for the 4404.

WHERE TO FIND INFORMATION

You have several important sources of information on the 4404:

- This manual, the 4404 Reference Manual, contains the syntax and details of commands and utilities. This manual also contains the details of the assembler, linking loader, 'C' compiler, and the remote terminal emulator.

- 4404 User's Manual The User's manual contains basic information on system installation, startup, installing software, and the other "how to put commands together" discussions. See the index of the User's manual to find how to perform particular tasks.

- The on-line "help" utility, which contains a very brief description of the syntax of user commands.

- The Introduction to Smalltalk-80(tm) manual, which contains details and a short tutorial on the Smalltalk-80 programming language.

- The reference manuals for the optional languages available on the 4404.

MANUAL SYNTAX CONVENTIONS

Throughout this manual, the 4404 User's Manual, and in the on-line Help files, the following syntax conventions apply:

1. Words standing alone on the command line are keywords. They are the words recognized by the system and should be typed exactly as shown.

2. Words enclosed by angle brackets ("<" and ">") enclose descriptions that you must replace with a specific argument. If an expression is enclosed only in angle brackets, it is an essential part of the command line. For example, in the line:

   addusr <user_name>

   you must specify the name of the user in place of the expression <user_name>.

3. Words or expressions surrounded by square brackets ("[" and "]") are optional. You may omit these words or expressions if you wish.

4. If the word "list" appears as part of a term, that term consists of one or more elements of the type described in the term, separated by spaces. For example:

   <file_name_list>

   consists of a series (one or more) of file names separated by spaces.

Section 2

USER COMMANDS AND UTILITIES

You can use the commands and utilities in this section from any user account. Some options, however, require special privileges. These options are mentioned in the detailed description of each command or utility.

asm

The "asm" command is the 68000/68010 relocating assembler.

SYNTAX

asm <file_name_list> [+befFlLnosStu]

DESCRIPTION

b	Suppress binary output.
e	Suppress summary information.
f	Disable field formatting.
F	Enable "fix" mode. (Comments that begin with a semicolon, ';', are assembled.)
l	Produce a listing of the assembled source.
L	Produce listing of input file during the first pass.
n	Produce decimal line numbers with the listing.
o=<file_name>	Specifies the name of the binary file.
s	Produce a listing of the symbol table.
S	Limit symbols internally to 8 characters.
u	Classify all unresolved symbols as external.

EXAMPLES

1. asm asmfile
2. asm test.a +euo=test.r
3. asm test.a test2.a test3.a +blns

The first example assembles the source file "asmfile" and produces the relocatable binary file "asmfile.r". The assembler sends summary information to standard output, but produces no source listing. Any errors detected are sent to standard output.

The second example assembles the file "test.a" and produces the relocatable file "test.r". No summary information is produced, and all unresolved references are classified as external. If the assembler detects no errors during the assembly, the user sees no output from this command.

The third example assembles the three files, but produces no binary output. A listing with a symbol table is sent to standard output. The listing includes decimal line numbers.

SEE ALSO

Section 5, The Assembler and Linking Loader

backup

Copy files from the file system to the floppy device.

SYNTAX

backup [+ AbBdlp] [+ a= days] [+ t [= filename]] [file ...]

DESCRIPTION

The "backup" command is used to create and maintain archival backups of files or directories on the system. It can operate in two distinct modes, selected by options: create mode, and append mode. Create mode copies the specified files or directories to the backup device, and destroys any data that is already on the backup device. Append mode adds the specified files or directories to existing files on the the backup device. Thus, it is possible to append, to an existing backup file, a file whose path and file names are identical with one already backed up.

The "backup" command stores files and directories on the flexible disk drive ("/dev/floppy"). The "backup" command uses a unique file structure, which is completely different from the standard operating system file structure. Therefore, "/dev/floppy" must not be mounted onto the file system using the "mount" command. The only way to read devices written by "backup" is to use "restore." The only other command that you should use on a backup device is "devcheck".

The backup disk should generally be formatted before the back up operation begins. Although the file structure created by the format command is destroyed by "backup", the raw track formatting is essential. During the back up process, you can request that "backup" format disks before writing to them by pressing "f" rather than "Return" when backup prompts you to "enter C/R."

Back ups may extend over more than one volume of the backup medium. There are no restrictions on the sizes of files copied. If necessary, "backup" breaks files into segments and stores each segment on a different volume.

Arguments

<file_name_list>	List of the names of files and directories to process. Default is the working directory.

If you specify a directory name as an argument in create or append mode, the program processes only the files within that directory. If you also specify the 'd' option, the program restores all files within the given directory and its subdirectories.

Options Available

a=<days>	Copy only those files that are no older than the specified number of days. A value of 0 specifies files created since midnight on the current day; a value of 1 specifies files created since midnight of the previous day, and so forth.
A	Append to a previous backup.
b	Print sizes of files in bytes.
B	Do not back up files that end in ".bak".
d	Back up entire directory structures.
l	List file names as they are copied.
p	Prompt you with each file name to determine whether or not the backup procedure should be performed on that particular file.
t[=<file_name>]	Back up only files that have been created or modified since the date in the specified file. When the backup is finished, update the date in the file (see NOTES). If you do not specify a file, the default is ".backup.time".

With no options, "backup" is quiet. The 'l' option allows you to see what the program is actually doing.

If you specify the 't' option, but the "backup time" file specified as its argument does not yet exist, "backup" copies all the files and directories listed on the command line. Thus, a user may obtain a full backup (either without the 't' option or with a nonexistent "backup time" file) or a partial backup, which includes only those files created since the last backup.

EXAMPLES

1. backup +l

2. backup +ld file1 file2 dir1 dir2

3. backup +ld file1 file2 dir1 dir2 +a=5

4. backup +lt

5. backup +lAt=backup_time

The first example backs up all files in the working directory to the device "/dev/floppy". The file names are listed as they are copied to the device.

The second example copies (in order) the files "file1" and "file2", then all files and directories contained in the directories "dir1" and "dir2".

The third example performs the same function as the second example, except that it copies only those files that are five days old or less.

The fourth example creates the same backup as the first example, but only copies the files created or modified after the time contained in the file ".backup.time". If this file does not exist, all the files are copied.

The fifth example adds a set of files to a previously created backup. In particular, it adds exactly the files that were created or modified since the creation of the file "backup_time".

NOTES

- When using append mode, you must place the final diskette in the backup device. Because the "backup" command always expects to receive the diskettes in order, it issues a message saying that you have inserted the wrong volume and prompts for permission to continue. In this case you want the last volume in the drive and should respond with a 'y' to the prompt. The program then appends files to that volume, requesting new volumes as necessary.

- When files are restored, they are generally restored to the same directory location as you specified when they were backed up. As files are backed up, "backup" makes an indication of the path name for each file. When files are restored, the program uses the path name to place the file in its proper directory location. If the path name is relative (i.e., does not begin with '/'), the path name of the restored directory is also relative. Thus, files backed up with a relative path name may be restored to a directory location different from the one in which they were created.

  An example should make this clear. If the working directory is backed up, either by specifying no source files or by using the directory name '.', the files are backed up with a relative path of '.'. When these files are restored, they are placed in the directory '.', which might not be the same directory they originally came from. This feature allows the manipulation of entire file systems in a general fashion. To specify a unique directory location for a file, you should specify its entire path name, starting with '/'.

MESSAGES

Several of the following messages prompt you for a positive or negative response. The program interprets any response that does not begin with an upper or lowercase 'n' as a positive response.

```
Backup to "<file_name>"
Update backup on "<file_name>"
```

These messages are printed when "backup" begins. They notify you of the function about to be performed.

```
Copy "<file_name>" (y/n)?
```

If you specify the 'p' option, the program prints this prompt before it takes any action. A response of 'n' or 'N' indicates that the operation should not be performed for the given file. Any other response is interpreted as "yes".

```
Device model name?
```

You should respond to this prompt with "TEK4404".

Do you wish to abort "append" function and create a new backup?

This message is printed at the initiation of the "append" operating mode if an invalid header (indicating a bad backup format) is detected. You have the option of aborting from "append" mode and switching to "create" mode.

Format program name?

This prompt is issued in response to a "format" request for the next volume. It indicates that the program could not find a format program name in the file "/etc/format.control." You should respond with "format" since you are backing up on the flexible disk drive.

Insert next volume - Hit C/R to continue:

This prompt is issued when the program needs a new backup disk. You should type a carriage return only when the next disk has been placed in the drive. When creating new backups or when appending to an old one, you may enter the character 'f', followed by a carriage return. If the program is in append mode, it automatically switches to create mode and starts a new backup. The 'f' indicates that the disk has been inserted in the drive, but that it must be formatted before continuing. In this case the program first checks the file "/etc/format.control" for a format program name, and if found formats the disk. If it cannot find this file, it then prompts you for the format program necessary to format the disk. Subsequent format operations use the same information; thus, all disks that were not previously formatted must have the same characteristics (e.g. double-sided, double-density).

The program prints these messages as it takes the corresponding action during a creation operation.

This is Volume #<number_1> -- Expected Volume #<number_2> -- Continue?

The program expects you to insert volumes in sequential order. If a volume appears out of order, "backup" prints this message. If you type anything except an 'n' or an 'N' as the first character of the response to the message, "backup" ignores the fact that the volumes are out of order and continues with the backup. Otherwise, it prompts you for another volume.

Volume name?

Each set of backup volumes has a name. You should enter the name "TEK4404" in response to this prompt. The name may contain as many as forty characters.

Volume <number> of "<vol_name>"

Whenever a new volume is inserted and properly validated, the program prints this message, which indicates the name of the backup volume and its sequence number.

ERROR MESSAGES

*** Invalid Volume Header -- Not a "backup" disk ***

The program validates each backup volume before using it. If this validation fails, the program prints this message to indicate that something is wrong. You then have another chance to insert the proper disk and continue. If validation fails while the program is in append mode, you may abort from append mode and create a totally new backup instead.

Write error! - file "<file_name>"

An I/O error occurred during the transfer of a file to the backup. An auxiliary message is printed indicating the nature of the error. The program tries to continue for all errors.

Unknown option: <char>

The option specified by <char> is not a valid option to the "backup" command.

** Warning: directory "<dir_name>" is too large!
** Some directories were ignored
**Warning: directory "<dir_name>" is too large!
** Some files were ignored

The program uses some internal tables during the back up process. If the limits of these tables are exceeded (highly unlikely), these messages are printed.

SEE ALSO

format
restore

cc

Invoke the 'C' compiler.

SYNTAX

cc <file_name_list> [+acDfiIlLmMnNoOqrRtUvwx]

where <file_name_list> is a list of the names of the files to compile.

Options Available

a	Produce as output assembly language source files with a ".a" extension.
c	Put comments in the assembly language file.
D<name>[=<defn>]	Command line "#define".
f	Produce an output module suitable for firmware.
I	Produce as output intermediate language files with a ".i" extension.
i=<dir_name>	Specify a directory for "#include" files.
l=<lib_name>	Specify a library name to be passed to the loader.
L	Produce a source listing and write it to standard output.
m	Produce load and module maps from the loader.
M	Leave the combined output as one ".r" file.
N	Produce a listing without expanding "#include" files.
n	Run the first pass only, do not produce any code.
O	Run the assembly language optimizer.

o=<file_name>	Specify the output file name.
q	Produce code that does calculations on "char" and "short" variables without first converting to "int".
R	Produce as output relocatable modules with a ".r" extension, and an executable module.
r	Produce as output relocatable modules with a ".r" extension.
s	Produce code that does not do stack growth checking.
S	Generate code that does do stack growth checking.
t	Produce a shared-text, executable output module.
U	Produce a line-feed character ($0A) for ' n' rather than the default of carriage return ($0D).
v	Show each phase of the compilation process (verbose mode).
w	Warn about duplicate "#define" statements.
x=<ldr_option>	Pass the information following the '=' on to the loader for processing.

For a full discussion of the 'C' compiler, refer to Section 7 of this manual.

chd

Change the user's working directory.

SYNTAX

chd [<dir_name>]

DESCRIPTION

The "chd" command, which is part of both the shell and script programs, changes the user's working directory to the directory specified on the command line. If no directory is specified, the default is the user's home directory (the directory entered on logging in). The user must have execute permission in the directory specified.

Arguments

<dir_name>	The name of the directory to use as the working directory. Default is the user's home directory.

EXAMPLES

1. chd /mark
2. chd book
3. chd

The first example changes the working directory to the directory "/mark".

The second example changes the working directory to the directory "book", which resides in the current working directory.

The third example changes the working directory to the user's home directory.

ERROR MESSAGES

Cannot change directories.

The operating system returned an error when the shell program tried to change directories. This message is preceded by an interpretation of the error produced by the operating system.

SEE ALSO

shell
script

commset

Set configuration of communications port.

SYNTAX

commset [options ...]

DESCRIPTION

This utility allows you to examine or set certain I/O options on the RS-232 communications port. With no argument, it reports the current setting of the options.

Options Available

The option strings are selected from the following set:

baud=nnn =external =nnn.mmm =default	Set the transmit and receive baud rates. Valid values are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 4800, 9600, 19200 and 38400. The keyword "external" specifies that the external clock should be used for the baud rate. The default of 9600 is used if the keyword "default" is entered. If two values are entered, then the first specifies the transmit rate and the second specifies the receive rate, otherwise both rates are set to the same value.
flag=dtr flag=input flag=output flag=inout flag=none flag=default	Set the type of flagging to be used. The keyword "dtr" specifies that the DTR and CTS signals should be used to flag input and output full conditions. The keywords "input" and "output" specify that CTL-S/CTL-Q flagging should be used for input and output, respectively. The keyword "tandem" specifies that CTL-S/CTL-Q flagging should be used for both input and output. The keyword "none" disables flagging. The default is inout flagging.

parity=even parity=odd parity=high parity=low parity=none parity=default	Set the type of parity to be used. The keyword "even" specifies that even parity be used. The keyword "odd" specifies that odd parity be used. The keyword "high" specifies that the parity bit should always be a one. The keyword "low" specifies that the parity bit should always be a zero. The keyword "none" specifies that the parity bit is treated as data. The default is low parity.
stop=n stop=default	Set the number of stop bits to be used. Valid values are 1 and 2. The default is one stop bit.
reset	Reset the communications port, flushing any pending data and setting all options to their default values.
show	Display the current settings for the options. This is the same as if no option is specified.

'C' IMPLEMENTATION NOTES

The "commset" command uses the "ttyset" and "ttyget" system calls to communicate option settings to the communications port device driver. The format of the 6-byte buffer used with these calls is defined differently than for standard tty devices. The include file "comm.h" contains definitions for the following structures and constants.

```
struct commbuf { char c_com, c_value, c_parity, c_flag, c_ospeed,
                 c_ispeed };
```

The c_com field is used to request various commands to be executed by the device driver during ttyset and ttyget calls. Valid values for this field are defined as follows:

RESET_COMM	1	Reset the communications port
SETUP_COMM	2	Set parity, flags and baud rates
EXCL_COMM	3	Do not accept open request until closed or reset
BREAK_COMM	4	Send break signal for c_value tenths of a second
NOBLOCK_COMM	5	Read calls do not block
BLOCK_COMM	6	Read calls do block (default)
DTRLOW_COMM	7	Set DTR signal low
DTRHIGH_COMM	8	Set DTR signal high (default)
RTSLOW_COMM	9	Set RTS signal low
RTSHIGH_COMM	10	Set RTS signal high (default)

The SETUP_COMM request causes the parity type and number of stop bits to be set according to the value in the c_parity field. Valid values for this field are defined as follows:

```
LOW_PARITY      0
HIGH_PARITY     1
EVEN_PARITY     2
ODD_PARITY      3
NO_PARITY       4
TWO_STOP_BITS   0x80    if msb set then two stop bits, else one
stop bit
```

The SETUO_COMM request also causes flagging to be set by the value in the c_flag field. Valid values for this field are defined as follows:

```
NO_FLAG 0
INPUT_FLAG      1
OUTPUT_FLAG     2
TANDEM_FLAG     3
DTR_FLAG        4
```

By default, read calls will block if no input is available. If any data is available, it is read into the caller's buffer (up to the requested number of bytes) and the number of bytes read is returned. If NOBLOCK_COMM is requested, then read calls will not block and a zero count is returned if no bytes are available.

The following constants are used in the c_ospeed and c_ispeed fields to indicate the transmit and receive baud rates:

```
EXTERNAL        0
     C50        1
     C75        2
     C110       3
     C134       4
     C150       5
     C300       6
     C600       7
     C1200      8
     C1800      9
     C2400      10
     C4800      11
     C9600      12
     C19200     13
     C38400     14
```

SEE ALSO

conset

compare

Compare two text files line by line and report the differences.

SYNTAX

compare <file_name_1> <file_name_2> [+<window_size>]

DESCRIPTION

The "compare" command compares two text files and indicates how they differ. The information provided is usually sufficient to allow the user to change one file into the other. By default, the "compare" command considers that it is in the same place in each of the files if three lines match.

The output from the command reports sets of lines which have been deleted from, added to, or changed in either file. These messages are written from the point of view of how to change the first file into the second file. For instance, the message

File "<file_name>" lines deleted

means that if the lines following the message are deleted from <file_name>, the two files will be the same.

The program also reports the presence of additional lines in a file with the following message:

File "<file_name>" has additional lines

This message is not from the point of view of changing one file into the other. Rather, it means that the file mentioned in the message is the file that contains additional lines.

If a set of lines is deleted from one file and the following line is changed as well, "compare" reports all those lines as lines that have been changed rather than inserted or deleted.

The "compare" command can handle files of any size, but can only process 250 lines at a time. If the files differ in any spot by 250 lines, the program reports 250 lines changed in each file and continues comparing them.

Arguments

<file_name_1>	The name of the first file to use.
<file_name_2>	The name of the file to compare to <file_name_1>

Options Available

<window_size>	Use the integer <window_size> as the number of matching lines required before considering the files synchronized. The number specified must be between 1 and 250. The default is 3.

EXAMPLES

1. compare /michael/test /cathy/test
1. compare test test.bak +5

The first example compares the file "test" in the directory "/michael" to the file "test" in the directory "/cathy".

The second example compares the two files "test" and "test.bak" in the working directory. The window size for the comparison is five lines.

ERROR MESSAGES

Syntax: compare <file_name_1> <file_name_2> [+<window_size>]

The "compare" command expects exactly two arguments. This message indicates that the argument count is wrong.

conset

Set or examine the configuration of the console port.

SYNTAX

conset [options ...]

DESCRIPTION

The utility "conset" allows you to examine and set certain I/O options on the console port. With no argument, it reports the current setting of the options.

Options Available

The option strings are selected from the following set:

+ raw - raw	Set or clear the raw mode.
+ echo - echo	Enable or disable character echoing.
+ tabs - tabs	Automatically expand tabs or don't.
+ becho - becho	Echo space/backspace to erase on backspace or don't.
+ schar - schar	Enable or disable single character mode.
+ xon - xon	Enable or disable ctrl-S/ctrl-Q flagging to suspend output.
+ any - any	Allow or don't allow any character to restart suspended output.
chardel= n	"n" is a hex number specifying a character to be used as the delete character.
linedel= n	"n" is a hex number specifying a character to be used as line delete character.
+ screensave - screensave	Enable or disable screen blanking after 10 minutes.
+ video - video	Normal video (black on white) or inverse video.

+ cursor - cursor	Make graphic cursor visible or invisible.
+ track - track	Enable or disable graphic cursor tracking the mouse.
+ mousepan - mousepan	Enable or disable mouse panning of the viewport.
+ diskpan - diskpan	Enable or disable joydisk panning of viewport.
show	Display the current settings for the options. This is the same as if no option is specified.

'C' IMPLEMENTATION NOTES

The conset command uses the "ttyset" and "ttyget" system calls to communicate the raw, echo, tabs, becho, schar, xon, any, chardel, and linedel option settings to the console port device driver and it uses system traps to implement the screensave, video, cursor, track, mousepan, and diskpan options.

SEE ALSO

commset

copy

Copy a file or directory to the specified file or directory, or copy one or more files to the specified directory.

SYNTAX

```
copy <file_name_1> <file_name_2> [+dbncotBplLDP]
copy <file_name_list> <dir_name> [+dbncotBplLDP]
copy <dir_name_1> <dir_name_2> [+dbncotBplLDP]
```

DESCRIPTION

Three forms of the "copy" command exist. The first form makes a copy of a file and gives it the specified name. The second form makes one copy of each specified file and places all copies in the specified directory. The last component of each file name is preserved in the new directory. The third form copies the contents of one directory to another.

In any case, if no file exists which has the same name as the name specified for the new copy, the "copy" command creates one. If a file with that name does already exist, it is deleted and recreated before copying takes place. Thus, the contents of the file are lost and replaced by the contents of the file being copied. In addition, any links to that file are broken.

The new file has the same permissions as the original file. The owner of the new file is always the user who executes the command. The user must have execute permission in the directory in which copies are to be made. He or she must also have write permission for the file being copied to and, unless the 'o' option is specified, in the directory that is to contain the new copy.

Arguments

<file_name_1> The name of the file to copy.

<file_name_2> The name of the new copy of the original file.

<file_name_list> A list of the names of the files to copy to the specified directory.

<dir_name> The name of the directory in which to place all copies.

Options Available

d Copy directory structure for all named directories.

b Do not copy a file unless it already exists in the destination directory.

n Copy a file if it is newer than the copy in the destination directory. If no copy exists, perform the copy.

c Do not copy a file if it already exists in the destination directory. Cannot be used with "n."

o Retain original file ownership.

t Don't create top level directories at destination.

B Don't copy files ending in ".bak".

p Prompt for permission to copy files.

l List the name of each file as it is copied and the name of the new copy.

L Don't unlink the destination file.

P Preserve the modification time of the source file.

EXAMPLES

1. copy parts parts.bak
2. copy letter /mark/letter +p
3. copy test_1 test_2 memo /mark +los

The first example copies the file named "parts" to a file named "parts.bak". If a file named "parts.bak" already exists, it is deleted and recreated before copying takes place.

The second example copies the file "letter" in the working directory to the file "/mark/letter". If a file named "/mark/letter" already exists, the "copy" command prompts for permission to alter its contents before proceeding. If the user denies permission, no copy is made. For the command to succeed the user must have both write and execute permission in the directory "/mark" as well as write permission for the file "/mark/letter".

The third example copies the files "test_1", "test_2", and "memo" to the directory "/mark". The last component of each file name is preserved in the new directory. Thus, the file specifications of the new files are "/mark/test_1", "/mark/test_2", and "/mark/memo". If a file with one of these names already exists, the "copy" command overwrites its contents without warning (the user does not need write permission in the directory "/mark"). The name of each file and the name of the new copy are listed as copying takes place. The command aborts immediately if it encounters an error (e.g., one of the files listed does not exist).

Each copy created by these commands has the same permissions as the original file. The owner of all copied files is the user executing the command.

ERROR MESSAGES

Entry does not exist: <file_name>

The user asked for a copy of a nonexistent file.

file_name_1> and <file_name_2> are the same file

A file may not be copied onto itself. Both <file_name_1> and <file_name_2> refer to the same file. (If their names are not the same, they are links to the same file.)

May not copy a directory: <dir_name>

The user asked for a copy of a directory. Directories may not be copied.

May not copy a special file: <file_name>

The user asked for a copy of a block or character file. Such files may not be copied.

Must be a directory: <file_name>

The form of the "copy" command being used requires the last argument to be an existing directory; <file_name> is not an existing directory.

Path cannot be followed: <file_name>

One or more of the directories which make up the name of the file do not exist.

Permissions deny access to file: <file_name>

The permissions associated with <file_name> or with the path leading to <file_name> prevent the user from accessing the file.

Read error on file: <file_name>

A physical read error occurred while reading <file_name>.

Syntax: copy <file_name_1> <file_name_2> [+lops]
copy <file_name_list> <dir_name> [+lops]

The "copy" command expects at least two arguments. This message indicates that the argument count is wrong.

Write error on file: <file_name>

A physical write error occurred while writing to <file_name>.

SEE ALSO

link
move
rename

crdir

Create a directory.

SYNTAX

crdir <dir_name_list>

DESCRIPTION

The "crdir" command creates a directory for each name listed as an argument to the command. The user must have write permission in the parent directory (the directory in which the new directory is created) of each directory created. Each new directory contains the entry ".", which represents the directory itself, and the entry "..", which represents its parent directory.

By default, "crdir" creates a directory with "rwxrwx" permissions. However, any default permissions set by the "dperm" command override these permissions. The owner may, of course, change the permissions at any time by using the "perms" command.

Arguments

dir_name_list> A list of the names of directories to create. All directories used in the name, except the last component of the name, must already exist.

EXAMPLES

1. crdir book
2. crdir /sarah/book

The first example creates the directory "book" in the working directory.

The second example creates the directory "book" in the directory "/sarah". If the directory "/sarah" does not already exist, the command fails.

ERROR MESSAGES

Error creating "<dir_name>": <reason>

The operating system returned an error when "crdir" tried to create the specified directory. This message is followed by an interpretation of the error returned by the operating system.

Error linking "<dir_name>" to its "." file": <reason>

The operating system returned an error when "crdir" tried to link the "." entry to the directory itself. This message is followed by an interpretation of the error returned by the operating system.

Error linking ".." to parent of "<dir_name>": <reason>

The operating system returned an error when "crdir" tried to link the newly created directory to its parent. This message is followed by an interpretation of the error returned by the operating system.

Error setting owner for "<dir_name>": <reason>

Initially, the "crdir" command creates the new directory with the owner "system". It then changes the owner to the user who executed the command. In this case, the operating system returned an error when "crdir" tried to change the owner of the directory. This message is followed by an interpretation of the error returned by the operating system.

Syntax: crdir <dir_name_list>

The "crdir" command expects at least one argument. This message indicates that the argument count is wrong.

SEE ALSO

dperm
perms
remove

create

Create an empty file for each file name on the command line.

SYNTAX

create <file_name_list>

DESCRIPTION

The "create" command creates an empty file for each name specified on the command line. If the file does not exist, it is created with "rw-rw-" permissions, and the owner is the user who executes the command. If the file already exists, the owner and permissions remain intact. However, the file is truncated to a length of 0.

Arguments

<file_name>	The name of the file to create. The last component of a file name may not contain more than fourteen characters. The "create" command ignores any additional characters.

EXAMPLES

1. create test
2. create /julie/test

The first example creates the file "test" in the user's working directory.

The second example creates the file "test" in the directory "/julie".

ERROR MESSAGES

Error creating "<file_name>": <reason>

The operating system returned an error when "create" tried to create <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Syntax: create <file_name_list>

The "create" command requires at least one argument. This message indicates that the argument count is wrong.

SEE ALSO

edit

date

Display or set the time and date.

SYNTAX

date [[<mm>-<dd>-<yy>] <hr>:<min>[:<sec>]] [+s}

DESCRIPTION

The "date" command has two forms: one with an argument and one without. Any user may execute the "date" command without an argument. In response, the system returns the current date and time. The user "system" may also use the "date" command with an argument to set the system date and time. If the user "system" uses the "+s" option, the system reads the hardware clock and sets the date and time accordingly.

Arguments

<mm> A number from 1 to 12 inclusive representing the month.

<dd> A number from 1 to 31 inclusive representing the day.

<yy> A two-digit number representing the last two digits of the year.

<hr> A number from 0 to 23 inclusive representing the hour. (Time must be expressed as 24-hour-clock time.)

<min> A number from 0 to 59 representing minutes past the hour.

<sec> A number from 0 to 59 representing seconds past the minute. The default is 0.

Options Available

s The "s" option allows the user "system" to set the system date for the internal hardware clock.

EXAMPLES

1. date 7-13-84 15:47:28
2. date 11:53
3. date 7-13 17:5
4. date
5. date +s

The first example sets the date to July 13, 1984, and the time to 3:47:28 P.M.

The second example sets the time to 11:53 A.M. The date defaults to the date stored in memory.

The third example sets the date to July 13 and the time to 5:05 P. M. The value for the year defaults to the stored value, and the value for seconds defaults to 0.

The fourth example displays the date and time currently stored in memory.

The fifth example sets the date and time to correspond to that in the system hardware clock.

ERROR MESSAGES

Invalid <arg> specified.

The value specified for the argument shown in the error message is not within the acceptable range.

Only the system manager may change the date!

The user who tried to change the date is not the system manager.

Syntax: date [[<mm>-<dd>-<yy>] <hr>:<min>[:<sec>]]

The syntax of the command line is incorrect. Most probably, the arguments specifying the time are missing.

debug

"debug" invokes a machine-language debugging system.

SYNTAX

debug [<image_file_name>]

DESCRIPTION

The "debug" command is used to aid in the testing and debugging of machine-language programs. Because all programs are ultimately translated into machine language, any program may be debugged using "debug."

The "debug" command is used to examine or modify the image of a machine-language program. This image can be (1) a post-mortem memory dump of a program which has been aborted by the operating system, (2) a program image file, or (3) a program which is currently executing under the control of "debug". If no image file is specified on the command line, the default is the file "core" in the working directory. The "debug" command examines the file to determine whether it is a "core" image or an executable image file. If it is neither, "debug" issues the message "Invalid image type" and terminates. The third type of image may be created only by specifying the name of an executable image on the command line, followed by executing 'x' command to create the controlled task.

The commands available with "debug" allow the user to examine memory locations within the program image, to modify memory locations, to set breakpoints, to execute single instructions (to single step through the program), to examine and change registers, and more. Some commands, such as single step, are applicable only when "debug" is being used to control the execution of a task. However, most commands are available for use with all image types.

Arguments

<image_file_name>	The name of the file to debug. The default is the file "core" in the working directory.

Commands Available

The "debug" command normally works in an interactive environment. The basic command structure is designed to be simple to use and to remember. In general, each command name is a single character, which may be followed by one or more expressions.

Expressions may include the operators '+' and '-', which are evaluated from left to right unless parentheses are used. Expressions may also include any of the following terms:

$<num>	The hexadecimal value of <num>.
<num>	The hexadecimal value of <num>. If this form is used, the number must start with a digit. If it starts with a character, "debug" interprets it as a symbol.
<num>	The decimal value of <num>.
<symbol>	The value of the specified symbol. Symbol names must be completely specified -- that is, all char characters are significant.
<register>	The contents of the specified register. The register may be D0 through D7, A0 through A7, SR, or PC. The letters used in specifying a register may be either upper- or lowercase. . The last memory address accessed.

"debug" includes these commands:

+	Execute a shell command.
=	Display the value of an expression in multiple formats.
?	Display the "help" menu.
b	Set a breakpoint.
B	List the breakpoints that are currently set.
c	Clear one or all breakpoints.

d	Dump a section of memory.
g	Continue execution of a program.
G	Execute the program until reaching a branch or a breakpoint.
i	Disassemble instructions.
I	Initialize symbol table.
k	Terminate the currently executing task.
K	Remove any pending signals for the controlled task.
m	Modify bytes in memory.
M	Display the current memory map.
n	Display the command line for the task.
q	Terminate "debug".
r	Display the contents of all registers.
R	Set the contents of a register.
s	Execute a single instruction.
S	Set a temporary breakpoint at the instruction following the current instruction and execute the current instruction.
T	Trace instructions until reaching a branch or a breakpoint.
x	Create a task to be executed under the control of "debug".

The following paragraphs describe "debug" commands in more detail:

+ <shell_command>

This command allows the user to execute a single shell command without exiting "debug".

= <expression>

This command displays the value of the expression symbolically, in hexadecimal, and in decimal.

?

This command displays a menu of commands available from "debug".

b <location> [<count>]

The 'b' command sets a breakpoint at the given location. When the program is executed, the instruction at the given location is replaced by a special instruction which indicates to the operating system that the user wants to break the flow of the program. When this instruction is executed in the program, the operating system suspends the program and notifies "debug", which prints the location of the breakpoint and returns to command mode. If the user specifies a count, the breakpoint is executed <count> times before execution is halted and "debug" notified. Once the count is exceeded, execution is halted every time the breakpoint is encountered unless it is reset by another 'b' command or cleared.

B

The 'B' command lists each breakpoint which is currently set as well as the corresponding <count> if it is nonzero.

c [<address>]

If the user does not specify an address, the 'c' command prompts for permission to clear all breakpoints that are currently set. If the user does specify an address, it clears the breakpoint at that address.

d <address_1> [<address_2_or_count>]

The 'd' command dumps the hexadecimal contents and the ASCII equivalents of a range of memory locations. Memory is displayed sixteen addresses to a line. Nonprintable characters are represented in ASCII by a period,

If the user specifies only one argument, the command displays the contents of the specified address. If the user specifies two arguments and the second one is greater than the first, the command interprets the second argument as an address. It displays the contents of memory from the first specified address to the second, inclusive. If the user specifies two arguments

and the second one is less than or equal to the first, the command interprets the second argument as a count. It displays the contents of memory beginning at the first address and continuing for the number of addresses specified by the second argument.

The dump may be aborted by typing the return key during the dump. Control-C does not abort the command.

g

The 'g' command continues the execution of a controlled task. Execution continues until the program terminates, receives a signal or encounters a breakpoint. The user may use this command only when executing a controlled task.

G

The 'G' command executes the program until it encounters any branch instruction, any call instruction, or any breakpoint.

i [<address_1> [<address_2_or_count>]]

The 'i' command displays the contents of memory from the first specified address to the second, inclusive. If the user specifies two arguments and the second one is less than or equal to the first, the command interprets the second argument as a count. The 'i' command interprets the specified location or range of locations as machine-language instructions and advances the location counter to the start of the last complete instruction within the specified range. If the user specifies no second argument or if the range specified by the second argument is shorter than the complete instruction, the command displays the instruction which begins at the starting address but does not move the location counter. A carriage return by itself is equivalent to the command "i .", except that the location counter is advanced to the beginning of the next instruction.

I

The 'I' command initializes debug's internal symbol table. The symbol table is used to interpret symbolic addresses and values. The 'I' command prompts for the name of the file containing the symbol table to use. The file must be a binary image file. This command is normally for use with a core image file, because such files do not contain any symbolic information. Once the symbol table is initialized, however, a core image file can be interpreted symbolically.

k

The 'k' command terminates execution of the current controlled task. If no controlled task exists, the command is not allowed. This command need not be used, because the 'x' command implicitly kills any controlled task before creating another.

K

When a task running under the control of "debug" receives a signal, the operating system notifies "debug" and suspends the task. The "debug" program then enters command mode, allowing the user to execute any "debug" command. A user who wishes to ignore the signal may do so by entering the 'K' command. A user who wishes the signal to take effect should simply continue the program with the 'g' (or a similar) command.

m <address>

he 'm' command modifies the contents of one or more memory locations in the image file. In response to this command, "debug" first displays the specified address and its contents. The user may change the contents by entering any expression, may leave the contents as is by entering a period, or may terminate the command by entering just a carriage return. Unless the user terminates the command, "debug" modifies the contents if appropriate, displays the next address with its contents, and waits for input from the user.

If the image file is a core dump or an executable file, the file itself is modified. If the image file is a controlled task (i.e., an 'x' command has been executed), only the memory of that task is altered. The executable file from which "debug" created the task is not changed. Therefore, when patching code the user should be aware that patches are applied only to the executing image file.

M

The 'M' command displays a map of the logical addresses available to the task image. If the image is either a core dump or a controlled task, the map contains the ranges of addresses being used by the program. These ranges may change whenever the program executes a "break" or a "stack" system call. If the image is an executable file, the 'M' command displays the ranges of the addresses of the TEXT and DATA/BSS segments.

n

The 'n' command displays the command line which was used to create the task. This is merely a display of the command arguments passed to the program when it was created. In most cases the command line consists of the shell command used to invoke the program. The command line for a controlled task looks just like the command line entered with the 'x' command that created it, except that the 'x' is replaced by the program name.

r

The 'r' command displays the contents of the registers for the image file, as well as the address of the program counter and the instruction located at that address. For a core dump it displays the contents of the registers at the time the program was aborted by the system and the location of the program counter at that time. The instruction displayed is the instruction that was in progress when the program was aborted. For a controlled task, the 'r' command displays the contents of the registers as they will be when execution resumes, the address at which execution will resume, and the instruction at that address. The registers for an executable file are undefined. For an executable file, the 'r' command displays the contents of the registers as zeros and the address and contents of the entry point of the program.

R <register_name> <expression>

The 'R' command, which may be used only if the image file is a controlled task, alters the contents of a register. The register may be D0 through D7, A0 through A7, SR, or PC. The letters used in specifying a register may be either upper- or lowercase. The supervisor portion (the upper byte) of the status register may not be altered.

s

The 's' command executes a single machine-language instruction. When the instruction is complete, "debug" displays the state of the task (including the new program counter) and the next instruction to be executed. The 's' command uses system facilities provided by the operating system. Thus, the user may safely single-step through macro operations such as system calls.

S

The 'S' command sets a temporary breakpoint at the instruction following the current instruction. This breakpoint is removed as soon as it is encountered. If another 'S' command is executed before the breakpoint is encountered, it removes the original breakpoint. This command may be used with any instruction, but it is normally used with a call to a subroutine.

T

The 'T' command executes the program until it encounters any branch instruction, any call instruction, or any breakpoint. After the execution of every instruction, "debug" displays the address of the next instruction and the instruction itself.

x [<arguments>] [<I/O_redirection>]

The 'x' command creates a controlled task from an image file. In order to execute this command, the user must first invoke "debug" with the name of an executable image file as the argument. The task is halted before execution of its first instruction, so that "debug" can accept commands to control its execution.

I/O redirection may be accomplished using the character '<' to redirect standard input, '>' to redirect standard output, and '%' to redirect standard error. No provisions are made for using either append mode (">>") or implied mapping (">%").

NOTE

The more breakpoints you set, the longer the program takes to execute.

ERROR MESSAGES

Breakpoint table full!

The user has already set the maximum number of breakpoints.

Can't access core/image "<image_file_name>"

The operating system returned an error when "debug" tried to access the specified file. Most probably, either the file does not exist or the user does not have read permission in the file.

Can't open "<file_name>"

The "debug" command was unable to open the file which the user specified as the file containing the symbol table to use. Most probably, either the file does not exist or the user does not have read permission in the file.

Can't write "<image_file_name>"

The user tried to use the 'm' command to modify the contents of a memory location in the image file, but "debug" was unable to write to the file. Most probably, the user does not have write permission in the file.

Command too complicated

The user tried to use the '+' command to execute a shell command from "debug", but the command line was too long for "debug" to interpret.

Error during EXEC - <error_num>

The operating system returned an error when the user tried to create a controlled subtask using the 'x' command. This message is followed by the error number returned by the operating system.

Error in expression

The expression used contains a syntax error.

Illegal address

The address specified is not in the user's address space.

Illegal command, <char>, - ignored

The command specified by <char> is not a valid command for "debug". The character is ignored, and "debug" prompts the user for another command.

Illegal file type

The 'I' command cannot determine the file type of the image file and, consequently, ignores the file. All previously defined symbols are no longer defined.

Illegal register name

The register name specified by the user is not a valid register name. The register name must be one of the following: D0 through D7, A0 through A7, SR, or PC. The letters used may be upper- or lowercase.

image_file_name>" is not executable

The user does not have execute permission in the specified image file.

Invalid image file "<file_name>"

The file specified to the "debug" command must be either an executable file or a core dump.

No command line

The file being debugged is not a core file, and was not invoked with the 'x' command. Therefore, no command line exists for the file.

Not executing a task!

The command specified can execute only if the user has previously executed the 'x' command.

Sorry, can't execute a "core" file

The 'x' command cannot be executed on a core file.

** Syntax error

The 'x' command cannot parse the specified command line.

Undefined symbol

An expression contains a term which appears to be a symbol (starts with a letter or an underscore character, '_') but is not in the symbol table. Hexadecimal values used in expressions must begin with a digit (a leading 0 is accepted) or a dollar sign,'$'.

dir

List either the contents of a directory or information about a file.

SYNTAX

dir [<file_name_list>] [+abdflrsSt]

DESCRIPTION

The "dir" command is used to list either the names of the files in the specified directory or, if the argument is not a directory, information about the specified file. By default, the names of the files in a directory are listed in alphabetical order with several names per line.

Format of the Output

The information given about a file is presented on one line, which contains several fields. These fields are described here in the order in which they appear.

<fdn_num> The number of the file descriptor node (fdn) which describes the file in question. This field is not present unless the user specifies the 'f' option.

<file_name> The name of the file being described.

<size> The size of the file in blocks. If the file is a device, "dir" places the major and minor device numbers in this field.

<file_type> A single character specifying the type of file. The character 'b' represents a block device; 'c', a character device; and 'd', a directory. If the field is blank, the file is a regular file.

<perms> This field, which is composed of six columns, indicates what permissions are associated with the file. The first three columns represent permissions for the user who owns the file; the last three for other users. Permissions are always presented in the order read, write, and execute. They are represented by the letters 'r', A hyphen in a column means that the corresponding permission is denied. For example, if the permission field contains the sequence "rwxr-x", the user who owns the file may read, write, and execute the file, whereas other users may only read and execute it.

<link_count> The link count is the number of directory entries which point to a file. The link count for a directory is always at least 2 because the "." entry within the directory itself points to the same fdn as the directory entry for that file in its parent directory.

<owner> The user name of the owner of the file.

<last_mod_time> The time and date at which the file was last modified.

Arguments

<file_name_list> A list of the names of files to process. The default is the working directory.

Options Available

a List all files in a directory, including those whose names begin with a period, '.'. This option has no effect if the specified file is not a directory.

b List the file size in bytes rather than blocks. This option implies the 'l' option.

d If the file being processed is a directory, list the names of all files it contains. Continue this process for all descendant directories. This option allows the user to see the entire directory structure.

f List the number of the file descriptor node for each file. This option implies the 'l' option.

l If the specified file is a directory, give detailed information about each file in the directory. This option has no effect if the specified file is not a directory because in such a case the information is automatically given.

r If the specified file is a directory, reverse the order in which the files would otherwise be listed.

s If the specified file is a directory, list one file name on each line. This option is useful for creating a file which contains the names of all the files in a directory.

S Print a summary of the information after listing all files.

t Sort files by the time of their most recent modification. By default, the most recently modified file is listed first.

EXAMPLES

1. dir +l
2. dir /jay +abdfS
3. dir memo +f
4. dir /marcy +rt
5. dir /marcy +s

The first example lists information about each file in the working directory (except those whose names begin with a period).

The second example lists information about all files, including those whose names start with a period, in the directory "/jay" (the 'f' and the 'b' option both imply the 'l' option). In addition, the command displays a list of the files in each subdirectory that is a descendant of "/jay". The information includes the fdn number of each file. The size of each file is shown in bytes. At the end of the output is a summary showing the total number of directories processed, the total number of nondirectory files processed, and the total number of blocks used by all the files.

The third example displays information about the file "memo" in the working directory. The information includes the fdn number of the file.

The fourth example lists the names of those files in the directory "/marcy" which do not begin with a period. The names are sorted by the time of the last modification with the sense of the sort reversed so that the most recently modified file is the last one in the list.

The fifth example lists the names of those files in the directory "/marcy" that do not begin with a period. One name appears on each line.

ERROR MESSAGES

Unknown option: <char>

The option specified by <char> is not a valid option to the "dir" command.

** Warning: directory "<dir_name>" is too large!
** Some directories were ignored

The "dir" command cannot process a file if the total number of directories in every directory between that file and the directory specified on the command line exceeds 50. In order to make the command succeed, the user should start at a lower point in the directory tree.

** Warning: directory "<dir_name>" is too large!
** Some files were ignored

The "dir" command cannot list more than 500 file names from a single directory. In order to make the command succeed, the user should split the offending directory into two or more directories.

dperm

Set the default permissions for the creation of files by the current shell program or by tasks generated by the current shell program.

SYNTAX

dperm [<perms_list>]

DESCRIPTION

Every time a user creates a file, the operating system assigns it a set of permission bits which determines whether the file's owner and other users may read, write, or execute the file. The permissions assigned depend on the command used to create the file. The editor, for example, creates all files with "rw-rw-" permissions, which allow the user who owns the file, as well as other users, to read and write, but not execute, the file. The default permission for "crdir" are "rwxrwx"; for "create", "rw-rw-"; for "makdev", "rw-r--".

The "dperm" command, which is part of the shell program, is used to set the default permissions for the creation of a file. It allows the user to instruct the system always to deny certain permissions, independent of how the file is created. It is possible to independently turn off any of the permission bits for the file's owner and other users. If the user specifies no arguments, the operating system removes the existing default permissions.

It is only possible to deny permissions with the "dperm" command. The "perms" command may be used to add permissions to individual files. "perms" overrides the defaults set by "dperm".

Arguments

<perms_list> A list defining the permission bits to be used as defaults.

Format for Arguments

<perms_list> The first character of an element in a permissions list specifies if the argument applies to the user who owns the file ('u') or to other users ('o'). The second character must be a minus sign, '-', which indicates that the following permissions are to be denied. The minus sign is followed by one, two, or three of the characters 'r', 'w', and 'x' (for read, write, and execute).

EXAMPLES

1. dperm o-rwx
2. dperm u-w o-wx
3. dperm

The first example sets the default permissions so that the operating system denies all permissions to other users whenever it creates a file.

The second example sets the default permissions so that the operating system denies write permission to the user who owns the file, and both write and execute permission to other users whenever it creates a file.

The third example removes all default permissions.

NOTE

The "dperm" command is only effective while the shell program under which it is invoked is running. The default permissions for files created by the login shell can be permanently altered by placing the appropriate command in the file ".login" in the user's home directory. This file is automatically executed each time the user logs in.

ERROR MESSAGES

Error in permissions specification.

The format of the permissions list is incorrect. Most likely, the user has specified a plus sign, '+', instead of a minus sign, or has used an invalid character.

SEE ALSO

perms

dump

Send both a hexadecimal and an ASCII listing of a file to standard output.

SYNTAX

```
dump <file_name> [+i]
dump [<file_name_list>]
```

DESCRIPTION

The "dump" command sends a hexadecimal and an ASCII listing of a file to standard output. The two versions of the file appear side by side. A line of output consists of the address in the file at which that line starts, the hexadecimal contents of the byte at that address and of the following fifteen bytes, and the sequence of characters represented by these bytes. A nonprintable character appears as a period, '.', in the ASCII part of the listing.

The user may interrupt the "dump" command at any time by typing a control-C. Normally, a control-C returns the user to the shell program. However, if the "dump" command is in interactive mode and is actually displaying information when the user types a control-C, "dump" stops the output and prompts for another address.

Arguments

<file_name> The name of the file to dump. The default is standard input.

Options Available

i Enter interactive mode. The 'i' option may be used only if exactly one file name appears on the command line. If the user specifies the 'i' option, the "dump" command prompts for the address at which to begin. The address is relative to the first byte in the file, whose address is 0. An address preceded by a period is a decimal address; otherwise it is a hexadecimal address. The user may specify a single address, a range of addresses (two addresses separated by a

hyphen, or an initial address and an offset (an address followed by either a comma or a space, followed by a number). In the first case, the "dump" command displays sixteen bytes of information, beginning with the specified address. In the second case, it displays all the bytes from the first to the second address inclusive. In the third case, it begins displaying bytes at the address specified and continues for as many bytes as the following number dictates.

EXAMPLES

1. dump memo /cynthia/letter
2. dump letter +i
3. dump testprog >test.dump

The first example sends both a hexadecimal and an ASCII listing of the file "memo", which is the working directory, and the file "letter", which is in the directory "/cynthia", to standard output.

The second example enters interactive mode and prompts the user for the address at which to begin the dumping the file "letter".

The third example sends a hexadecimal and ASCII listing of the file testprog via redirected I/O to the file test.dump.

ERROR MESSAGES

Cannot interactively dump multiple files.

The 'i' option may not be used if more than one file name appears on the command line.

Cannot interactively dump standard input.

If the user specifies no file name on the command line, the default is standard input. The 'i' option may not be used in such a case.

Error opening "<file_name>": <reason>

The operating system returned an error when "dump" tried open <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Invalid option '<char>': ignored.

The option specified by <char> is not a valid option to the "dump" command. The command ignores it.

echo

Write the arguments on the command line to standard output.

SYNTAX

echo [<arg_list>] [+l]

DESCRIPTION

The "echo" command writes the arguments in <arg_list> to standard output. A space character appears after each string argument; no space appears after a hexadecimal argument; while the last argument is followed by a carriage return. You can use "echo" to non-destructively show how the "shell" or "script" programs evaluate special characters in the <arg_list>.

Arguments

arg_list> A list of arguments to write to standard output.

Format for Arguments

<arg_list> Each element in <arg_list> consists either of a string or of a hexadecimal number preceded by a plus sign, '+'.

Options Available

l Do not write a carriage return after echoing the argument list.

EXAMPLES

1. echo This is a test!
2. echo This is a test! +7 +l >/dev/console

The first example writes the string "This is a test!" to standard output, which defaults to the user's terminal.

The second example writes the string "This is a test!", followed by a bell character (hexadecimal 7), to standard output. Standard output is /dev/console (the 4404 display). The output is not followed with a carriage return. (The "+l" is the option "plus el", not the hexadecimal argument "plus one".)

edit

Invoke the text editor in order to create a new text file or edit an existing one.

SYNTAX

edit [<file_name_1> [<file_name_2>]] [+bny]

DESCRIPTION

The "edit" command may be used with zero, one, or two arguments. With one argument, "edit" opens the specified file for editing, creating it if necessary, and reads as much of the file as possible into the edit buffer. At the end of an editing session of a pre-existing file, the editor renames the original file by appending the letters ".bak" to its name. If this addition would result in a file name of more than fourteen characters (the maximum allowed by the operating system), the editor shortens the original name before adding the suffix. If a backup file already exists, the editor prompts for permission to delete it.

If the user specifies no arguments, the editor prompts for the name of the file at the end of the editing session, before returning control to the operating system. It does not accept the name of an existing file.

If the user specifies two file names, the operating system makes a copy of the first file specified, gives it the name specified by the second argument, and opens it for editing. If a file with that name already exists, the editor prompts for permission to delete it before proceeding. In such a case, the editor creates the new file with the same permissions as the old file.

Files created by the editor have permissions of "rw-rw-."

Arguments

<file_name_1> The name of the file to open for editing, or, if two file names are specified, the name of the file to copy.

<file_name_2> The name to give to the copy of the file specified by <file_name_1>. It is this copy that is opened for editing.

Options Available

b Do not save the original copy of the file as a backup file at the end of the editing session.

n Do not read any text into the edit buffer. This option allows the user to make large insertions at the beginning of a file.

y If only one argument appears on the command line, at end of the editing session automatically replace any existing backup file with the original copy of the file being edited. If two arguments appear on the command line and the second file specified already exists, delete that file at the beginning of the editing session.

EXAMPLES

1. edit test +ny
2. edit test oldtest

The first example opens the file "test" in the working directory but does not read any of it into the edit buffer. If the file does not exist, the editor creates it. At the end of the session, "edit" automatically replaces any existing backup file with the original copy of "test".

The second example makes a copy of the file "test", names it "oldtest", and opens it for editing. If a file named "oldtest" already exists, the editor asks for permission to delete it.

MESSAGES

Delete existing copy of new file?

The file specified by <file_name_2> already exists. If the user responds with a 'y', the editor deletes the existing copy of the file and opens the new file for editing. If the user responds with an 'n', the editor leaves the existing file intact and returns the user to the operating system.

File already exists
File name?

The "edit" command was executed with no arguments on the command line. At the end of the editing session, when the editor prompted for the name of the file, the user specified an existing file. Under these circumstances, the editor does not accept the name of an existing file.

ERROR MESSAGES

Cannot create new file

The editor cannot open the file specified by <file_name_2>. Most probably, either the user specified a path name that could not be followed or the user does not have the permissions necessary to open the file.

Cannot open edit file

The editor cannot open the file specified by <file_name_1>. Most probably, either the user specified a path name that could not be followed or the user does not have the permissions necessary to open the file.

Cannot read edit file

The editor encountered an I/O error trying to read the specified file.

Edit file does not exist

The user has specified two file names on the command line, but <file_name_1> does not exist.

New file is the same as the old file

Both <file_name_1> and <file_name_2> refer to the same file. (If their names are not the same, they are links to the same file.)

Too many file names specified.

The "edit" command requires zero, one, or two arguments. This message indicates that the argument count is wrong.

Unknown option specified

An option on the command line is not a valid option to the "edit" command. The command ignores the option and proceeds.

SEE ALSO

dperm
Section 9, EDIT, The Text Editor

find

Search for a string in a file or in standard input.

SYNTAX

```
find [+cu] <str_1>[&<str_2>] [<file_name_list>]
```

DESCRIPTION

The "find" command looks in the specified file for the specified string. By default, lowercase characters and uppercase characters are distinct.

Arguments

<str_1> The string to search for.

<str_2> The second string to search for (only if '&', the "and" operator, is used).

<file_name_list> A list of the names of files to search. The default is standard input.

Specifying a String

The user may completely specify a string or may take advantage of the matching characters recognized by the "find" command. Because some of these matching characters also have special meanings to the shell program, strings which use them must be enclosed in single or double quotation marks.

\ When used just before any matching character, including itself, the backslash character negates the matching ability of the character.

? The question mark matches any character except a new-line character.

< A left angle bracket specifies that the following string must be found at the beginning of a line. It loses its matching ability if it is not the first character of the string.

> A right angle bracket specifies that the preceding string must be found at the end of a line. It loses its matching ability if it is not the last character of the string.

& The "and" operator may be used between two strings (see the syntax statement). The "find" command reports only those lines on which both strings occur.

[] Square brackets enclose a list or a range of characters from which the "find" command can choose when looking for a string. A list of characters consists of adjacent characters. A range consists of two characters separated by a hyphen.

! The exclamation point may be used in conjunction with the square brackets. If it is the first character inside the brackets, the "find" command can choose from all characters not specified in the brackets when looking for a string.

Options Available

Any options used with the "find" command must appear immediately after the command name.

c Instead of writing the lines that contain the specified string to standard output, report the number of lines containing the string.

u Do not distinguish between upper- and lowercase.

EXAMPLES

```
1. find +u syntax test
2. find +u "<syntax" test trial
3. find +u 'syntax&statement' test
4. find +c " <" test
5. find +u '[a-e]nd' test
```

The first example writes to standard output all lines from the file "test" which contain the string "syntax". The command does not distinguish between upper- and lowercase.

The second example writes to standard output all lines from the files "test" and "trial" which contain the string "syntax" at the beginning of the line. The command does not distinguish between upper- and lowercase. Because matching characters are used to specify the string, the string must be enclosed in either single or double quotation marks.

The third example writes to standard output all lines from the file "test" which contain both the string "syntax" and the string "statement".

The fourth example writes to standard output the number of lines in the file "test" which contain a left-hand angle bracket. The matching ability of the angle bracket is negated because of the backslash character which precedes it.

The fifth example writes to standard output all lines from the file "test" which contain any of the following strings: "and", "bnd", "cnd", "dnd", or "end".

ERROR MESSAGES

Error opening "<file_name>": <reason>

The operating system returned an error when "find" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error processing "<file_name>": <reason>

The operating system returned an error when "find" tried to process the specified file. This message is followed by an interpretation of the error returned by the operating system.

Invalid option: '<char>'. Command aborted.

The option specified by <char> is not a valid option to the "find" command.

Syntax: find [+cu] <str_1>[&<str_2>] [<file_name_list>]

The "find" command expects at least one argument. This message indicates that the argument count is wrong.

SEE ALSO

shell
script

format

Format a flexible disk for use on the 4404 flexible disk drive.

SYNTAX

format [+fFqv]

Options Available

f=<blocks> Establish <blocks> blocks for file descriptor nodes (fdns).

F Logical format only. No physical format performed.

q Use quiet mode.

v Verify the disk after formatting.

DESCRIPTION

The "format" command formats a flexible disk for use in the 4404's flexible disk drive, "/dev/floppy." The device model name is "TEK4404" which formats the disks as double-sided, double-density, 40 TPI, with eight 512-bit sectors per track.

DETAILED DESCRIPTION OF "FORMAT" OPTIONS

The 'f' Option

Formatted disks use fdn blocks (each fdn block contains eight fdns) to hold information about files on the disk. By default, "format" uses 3% of the total disk space for fdn blocks. You can ovveride this default value with the 'f' option and specify the decimal number of fdn blocks to establish on the disk. At least one block must be allocated for fdns on every formatted disk.

The 'F' Option

The 'F' option does not physically format the disk. It performs a logical format only and erases all data on the disk.

The 'q' Option

Before actually starting to format the disk, "format" normally sends a prompt to ask if the user is ready to continue. The 'q' (quiet) option suppresses this prompt and inhibits all informative messages from "format" if no errors are encountered during formatting.

The 'v' Option

The 'v' (verify) option instructs "format" to verify the media after formatting. If this option is specified, "format" individually verifies every sector on the disk. It first writes an arbitrary pattern to each sector; then reads and verifies each one. Because verification of a large disk may take a long time, the "format" command prints symbols to indicate its progress. It prints an asterisk, '*', each time it finishes writing fifty sectors; a dollar sign, '$', each time it finishes reading and verifying fifty sectors. It reports any sectors which fail this test to the user.

The option is often desirable when the user is formatting a floppy disk because floppies do not automatically verify all written data.

free

Report the amount of free space available on the specified devices.

SYNTAX

free <dev_name_list> [+d]

DESCRIPTION

The "free" command reports the amount of free space remaining on the specified device. It reports both the total number of free blocks available for use in files and the total number of file descriptor nodes (fdns) available. The number of fdns available tells the user how many more files can be created on the device (assuming that sufficient free blocks remain for use in the files). If the number of available fdns drops to 0, no more files can be created on the disk, no matter how many free blocks remain.

Arguments

<dev_name_list>	A list of the names of the devices to report on. The devices may be either mounted or unmounted.

Options Available

d Provide more detailed information with the output. This extra information includes the names of the file system and the volume if they were specified when the disk was formatted, as well as the amount of swap space on the disk.

EXAMPLES

1. free /dev/disk

2. free /dev/floppy

The first example reports both the number of fdns available and the number of free blocks on the standard winchester hard disk.

The second example reports the same information on a mounted flexible disk.

ERROR MESSAGES

Cannot open <dev_name>

The specified device does not exist; the specified device exists, but no hardware is connected to it; or the device exists and hardware is connected to it, but no disk is in the device.

<dev_name> is not a block device.

The specified device must be a block device.

Unknown option: <char>

The option specified is not a valid option to the "free" command.

headset

Change information in the binary header of an executable file.

SYNTAX

headset <file_name_list> [+aAbBcCdSt]

DESCRIPTION

The "headset" command can alter certain portions of the binary header of an executable object module. Features such as whether or not the module is shared-text, whether or not the module can produce a core dump, and the initial stack size can be altered without reloading the module.

The characters used for options are identical to those used when invoking the loader with the "load" command. Those options which do not take an argument can be disabled by preceding the character with a minus sign, '-', instead of the usual plus sign, '+'.

Arguments

<file_name_list>	A list of the names of the files to process.

Options Available

a=<num>	Specifies the minimum number of pages to allocate to this task at all times. The minimum value for the argument is 0; the maximum, 32767. The default is 0. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs.
A=<num>	Specifies the maximum number of pages to allocate to this task at all times. The minimum value for the argument is 0; the maximum, 32767. The default is 0. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs.

b=<task_size> Specifies the maximum size to which the task may grow. The argument <task_size> may be "128K", "512K", "2048K", "8192K", "2M", or "8M". The default is "128K". The letters 'M' and 'K' can be either upper- or lowercase.

If the task size specified by the user is not large enough to hold the code from all the modules being loaded, "headset" automatically adjusts the size to the smallest value that can contain all the code.

Set a bit in the binary header of the output module which tells the operating system to zero neither the bss space nor any memory allocated while the task is running.

c=<source_type> Sets a flag in the binary header of the output module which indicates the type of source code from which the module was created. The argument <source_type> may be "ASSEMBLER" or "C". The names can be specified in either upper- or lowercase.

C=<config_num> By default, the loader uses the configuration number of the current hardware. The user may, however, use the 'C' option to specify a configuration number which overrides the default. This option is useful when loading a module for a machine other than the one on which it is running.

Set the "no core dump" bit in the binary header.

S=<hex_num> Specifies the initial stack size, which is written into the binary header of the module produced by the loader. The hexadecimal number is the number of bytes to reserve. The default is 0, in which case the system assigns the default stack size of 4K.

Produce a shared-text executable module.

EXAMPLES

1. headset mathtest +t -d +S=2000

2. headset run_1 run_2 +tB +a=10

The first example makes the executable object module "mathtest" a shared-text module. It turns off the "no core dump" bit, so that the program can produce core dumps, and sets the initial stack size to hexadecimal 2000.

The second example changes the headers in the files "run_1" and "run_2". Both modules become shared-text modules. The operating system will zero neither the bss space nor any memory allocated while the task is running. The minimum page allocation is set to ten pages.

NOTES

- The user may make a change in a header which results in an inconsistent header. In such a case the "headset" command makes whatever adjustments are necessary in the fields which were not changed to remove the inconsistency. The user is notified of these adjustments.

- For example, if the user alters the initial stack size, the task size might have to be changed. If this change is necessary, "headset" notifies the user and adjusts the task size to the appropriate value. Adjustments may also be made when either the minimum or maximum page allocation is altered.

- If the task size specified by the user is not large enough to hold the code from all the modules being loaded, "headset" automatically adjusts the size to the smallest value that can contain all the code.

- If the user changes either the minimum or the maximum value for page allocation so that the minimum is greater than the maximum, "headset" automatically adjusts them according to the following rules.

    - The value for the maximum is always greater than or equal to the value for the minimum.

    - The value for the maximum can be 0, but if it is greater than 0, it must be at least 4.

MESSAGES

File "<file_name>": changed max page allocation to <num>.

The user specified a minimum page allocation that was above the current maximum page allocation. The utility set the maximum equal to the minimum.

File "<file_name>": changed min page allocation to <num>.

The user specified a maximum page allocation that was below the current minimum page allocation. The utility set the minimum equal to the maximum.

File "<file_name>": task size set to <task_size>.

The "headset" command had to adjust the task size either because the user specified an initial stack size that made the module larger, or because the task size specified on the command was too small for the calculated size of the module.

ERROR MESSAGES

Error opening "<file_name>": <reason>

The operating system returned an error when "headset" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error processing "<file_name>": <reason>

The operating system returned an error when "headset" tried to process the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>": <reason>

The operating system returned an error when "headset" tried to read the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error seeking in "<file_name>": <reason>

The operating system returned an error when "headset" tried to seek in the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error writing to "<file_name>": <reason>

The operating system returned an error when "headset" tried to write to the specified file. This message is followed by an interpretation of the error returned by the operating system.

File "<file_name>" is not a binary file.

The specified file does not contain a binary header.

File "<file_name>" is not a regular file.

The specified file is either a device or a directory.

File "<file_name>" is not executable.

The specified file is not an executable binary file.

Illegal configuration specified.

The configuration type must be between 0 and 255 inclusive.

Illegal hex number: <hex_num>.

The number specified is not a valid hexadecimal number.

Illegal maximum page allocation specified.

The maximum page allocation must be between 0 and 32767 inclusive.

Illegal minimum page allocation specified.

The minimum page allocation must be between 0 and 32767 inclusive.

Illegal task size specified.

The argument specified is not a valid argument to the 'b' option.

Invalid option: '<char>'.

The option specified by <char> is not a valid option to the "headset" command.

Minimum page allocation greater than maximum.

Both the 'a' and 'A' options appeared on the command line, but the minimum page allocation specified was greater than the maximum.

Unknown source type specified.

The argument specified is not a valid argument to the 'c' option.

help

Display a brief description of the use and syntax of the specified command.

SYNTAX

help [<command_name_list>]

DESCRIPTION

The "help" command displays a brief description of the use and syntax of the specified command. To obtain this information, it looks for a file in the "/gen/help" directory with the same name as the specified command. Descriptions of most 4404 commands are available. If you enter "help help" or "help" with no arguments, the "help" command displays a list of all the commands it can help with and prompts for the name of a specific command. Typing a carriage return terminates the command.

Arguments

<command_name_list> A list of the names of commands about which the user wants information.

EXAMPLES

1. help copy remove
2. help

The first example displays brief descriptions of the use and syntax of the "copy" and "remove" commands.

The second example displays a list of all the commands that the "help" command can help with, followed by a prompt for the name of a specific command.

NOTES

- The user may add files to "/gen/help". When the "help" command is executed, it simply looks for the specified file in "/gen/help", reads the contents, and writes it to standard output.
- If the file specified is a directory, the "help" command lists the contents of the directory and asks what command the user would like help with. If the command specified is not in that directory, "help" prompts for permission to search "/gen/help".

ERROR MESSAGES

Cannot help with <command_name>.

No description of the specified command is available to the "help" command.

Error opening "<file_name>": <reason>

The operating system returned an error when "help" tried to open the file <file_name>, which describes the specified command. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>": <reason>

The operating system returned an error when "help" tried to read the file <file_name>, which describes the specified command. This message is followed by an interpretation of the error returned by the operating system.

Too many files in directory.

The "help" command cannot function if the directory "/gen/help" contains more than 500 entries.

info

Display the contents of the information field associated with the specified binary file.

SYNTAX

info <file_name_list>

DESCRIPTION

A binary file may have an "information field" that stores textual information associated with the file. This information can include things like the version number and release date of the file, as well as other useful information pertaining to the file. The "info" command displays the contents of the information field.

Arguments

<file_name_list>	A list of the names of the files for which to display the information field.

EXAMPLES

1. info /system.boot
2. info /bin/edit /bin/info

The first example displays the version number, release date, and copyright information for the file "/system.boot", the operating system itself.

The second example displays version numbers, release dates, and copyright information for the text editor ("/bin/edit") and the "info" command ("/bin/info").

ERROR MESSAGES

Error opening "<file_name>": <reason>

The operating system returned an error when "info" tried to open the file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error processing "<file_name>": <reason>

The operating system returned an error when "info" tried to process the file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>": <reason>

The operating system returned an error when "info" tried to read the file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error seeking in "<file_name>": <reason>

The operating system returned an error when "info" tried to seek to the appropriate location in <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error writing to "standard output": <reason>

The operating system returned an error when "info" tried to write the output of the "info" command to standard output. This message is followed by an interpretation of the error returned by the operating system.

file_name>" has no information field.

The optional information field is not present in the specified file.

file_name>" is not a binary file.

The specified file lacks the header which identifies it as a binary file. The argument to the "info" command must be a binary file.

file_name>" is not a regular file.

The specified file is a directory or a special file (a block or character device). The argument to the "info" command must be a regular file.

Syntax: info <file_name_list>

The "info" command requires at least one argument. This message indicates that the argument count is wrong.

SEE ALSO

addinfo
Section 5, The Assembler and Linking Loader

int

Send a program interrupt to another task.

SYNTAX

int <task_ID> [+<int_num>]

DESCRIPTION

The "int" command sends the specified interrupt to the task identified by the task ID on the command line. If the user does not specify an interrupt, a termination interrupt (SIGTERM) is sent. A task ID is reported by the shell program whenever the user executes a task in the background. An ID can also be determined by the "jobs" command.

Arguments

<task_ID> The task ID of the task to interrupt. A task ID of 0 specifies all tasks associated with the user's terminal and owned by the user. +<int_num> The number associated with the interrupt the user wishes to send. The plus sign, '+', is necessary to distinguish the number of the interrupt from the task ID. Table 2-1 shows a list of the possible interrupts.

Table 2-1

POSSIBLE INTERRUPTS

Name	Number	Description	A C D I R
SIGHUP	1	Hangup	+ + - + +
SIGINT	2	Keyboard	+ + - + +
SIGQUIT	3	Quit	+ + + + +
SIGEMT	4	EMT $Axxx emulation	+ + + + +
SIGKILL	5	Task kill	+ - - - +
SIGPIPE	6	Broken pipe	+ + - + +

SIGSWAP	7	Swap error	+ - - - +
SIGTRACE	8	Trace	+ + - + -
SIGTIME	9	Time limit	+ + + - +
SIGALRM	10	Alarm	+ + - + +
SIGTERM	11	Task terminate	+ + - + +
SIGTRAPV	12	TRAPV instruction	+ + + + +
SIGCHK	13	CHK instruction	+ + + + +
SIGEMT2	14	EMT $Fxxx emulation	+ + + + +
SIGTRAP1	15	TRAP #1 instruction	+ + + + +
SIGTRAP2	16	TRAP #2 instruction	+ + + + +
SIGTRAP3	17	TRAP #3 instruction	+ + + + +
SIGTRAP4	18	TRAP #4 instruction	+ + + + +
SIGTRAP5	19	TRAP #5 instruction	+ + + + +
SIGTRAP6	20	TRAP #6-14 instruction	+ + + + +
SIGPAR	21	Parity error	+ - + - +
SIGILL	22	Illegal instruction	+ - + - +
SIGDIV	23	DIVIDE by 0	+ + + + +
SIGPRIV	24	Privileged instruction	+ - + - +
SIGADDR	25	Address error	+ - + - +
SIGDEAD	26	Dead child	- + - + +
SIGWRIT	27	Write to READ-ONLY memory	+ - + - +
SIGEXEC	28	Execute from STACK/DATA space	+ - + - +
SIGBND	29	Segmentation violation	+ + + - +

SIGUSR1	30	User-defined interrupt #1	+ + - + +
SIGUSR2	31	User-defined interrupt #2	+ + - + +
SIGUSR3	32	User-defined interrupt #3	+ + - + +
	33-63	Vendor-defined interrupts	

Notes

A = Default state is "abort" (otherwise, "ignore")
C = Interrupt can be caught
D = Produces a core dump
I = Interrupt can be ignored
R = Resets to default state when triggered

EXAMPLES

1. int 263
2. int +5 149
3. int 149 +5

The first example sends a termination interrupt (SIGTERM) to task number 263.

The second example sends a SIGKILL interrupt to task 149. No program can trap or ignore a SIGKILL interrupt.

The third example is identical to the second one. The order of the arguments is irrelevant.

ERROR MESSAGES

Error sending interrupt: <reason>

The operating system returned an error when "int" tried to send the interrupt. This message is followed by an interpretation of the error returned by the operating system.

Illegal interrupt specified: <int_num>

The number specified must be an integer between 1 and the number of signals, inclusive. At the time of this writing the number of signals is 32.

Illegal task ID specified: <task_ID>

The task ID specified contains some characters that are not digits. A legal task ID contains only digits.

Syntax: int <task_ID> [+<int_num>]

The "int" command expects exactly one task ID and no more than one interrupt number. This message indicates that the argument count is wrong.

SEE ALSO

jobs

jobs

Report the task IDs and starting times of all background tasks originated by the user from the current shell program.

SYNTAX

jobs

DESCRIPTION

The "jobs" command, which is part of the shell program, reports the task IDs and starting times of all background tasks originated by the user from the current shell program. (If "script" is running as the shell, the task IDs are preceded by the letter 'T' for task. This letter is not part of the task ID.)

EXAMPLES

jobs

This example is the only valid form of the "jobs" command. It reports the task ID and starting time of all active background tasks originated by the user from the current shell program.

MESSAGES

No tasks active.

The user has no active tasks in the background.

SEE ALSO

int

libgen

Create a new library or update an existing one.

SYNTAX

libgen o=<old_lib> n=<new_lib> [u=<update>] [<del_list>] [+al]

DESCRIPTION

The "libgen" command creates a new library of relocatable or executable modules or updates an existing library. Each module in a library must have a name. The name is assigned to a module by either the "name" pseudo-op in the relocating assembler or the 'N' option of the linking loader. The "libgen" command does not accept a module without a name.

As it runs, "libgen" produces a report describing the action that it takes for each module in the library. The report includes the name of the module and the file from which it was read (the old library or one of the update files).

Arguments

o=<old_lib> — The name of an existing library file that was previously created by the "libgen" command. "libgen" is being called to update an existing library rather than to create a new one. Either the "o=<old_lib>" or "n=<new_lib>" argument, or both, must appear on the command line.

n=<new_lib> — The name of a new library. If a file with this name already exists, "libgen" deletes it without warning before writing the new library. If the user does not specify a name for the new library, it defaults to the name of the old library. In such a case "libgen" puts the new library in a scratch file, deletes the old library, and renames the scratch file with the name of the old library. Either the "o=<old_lib>" or "n=<new_lib>" argument, or both, must appear on the command line.

u=<update>	The name of a file containing modules to add to the library. Modules of the same name are replaced by modules from the update file. The user may specify up to nine update files by repeating the "u=<update>" argument for each one.
del_list>	A list of the names of modules to delete from the old library.

Options Available

a Produce an abbreviated report that contains information only about modules that were replaced, added, or deleted.

l Suppress the production of a report.

EXAMPLES

1. libgen n=binlib u=one u=two u=three
2. libgen o=binlib u=new +a
3. libgen o=binlib u=newmods n=newlib transpose add +l

The first example creates a new library named "binlib" that contains all the modules from the files "one," "two," and "three."

The second example updates the library "binlib" by adding or replacing modules from the file "new." The command produces an abbreviated report.

The third example updates the library "binlib" by adding or replacing modules from the file "newmods" and by deleting the modules named "transpose" and "add". The updated library is written to the file "newlib". The old library is deleted.

ERROR MESSAGES

An old or new library name must be specified.

Either the "o=<old_lib>" or "n=<new_lib>" argument, or both, must appear on the command line.

No index found in <lib_name>

The "libgen" command creates every library with an index. This message indicates either that the file specified is not a library or that it is a library, but has been badly damaged, and can no longer be used.

Record not found in <module_name>

One of the files in the list of modules to delete from the old library was not found in that library. The command ignores that file name and continues.

Record with no name found in <module_name>

Every relocatable or executable module that goes into a library must have a name. The user should remake the specified module and give it a name.

Unknown argument: <str>

The argument specified by <str> is not a valid argument to the "libgen" command.

Unrecognizable record in <module_name>

All modules in a library must be either executable or relocatable.

SEE ALSO

ISection 5, The Assembler and Linking Loader
libinfo

libinfo

Display information about a library.

SYNTAX

libinfo <library_name_list> [+emM]

DESCRIPTION

The "libinfo" command lists the entry points and module names contained in a library produced by the "libgen" command. The user can optionally display only the entry points or only the module names. Information about a particular module within a library can also be displayed.

Arguments

<library_name_list>	A list of the names of the libraries to report on.

Options Available

e Display only entry points in the specified library.

m Display only module names in the specified library.

M=<mod_name> Display information about module <mod_name>. This option is incompatible with both the 'e' and 'm' options. If the user specifies incompatible options, "libinfo" uses the 'M' option and ignores any others.

EXAMPLES

1. libinfo testlib
2. libinfo runlib +m
3. libinfo /lib/mathlib +M=Arctan

The first example lists all entry points and module names in the library "testlib."

The second example lists all the module names contained in the library "runlib."

The third example displays the entry points and module names in the module "Arctan" in the library "/lib/mathlib."

ERROR MESSAGES

Error opening "<file_name>" : <reason>

The operating system returned an error when "libinfo" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>" : <reason>

The operating system returned an error when "libinfo" tried to read the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error seeking to <location> in "<file_name>" : <reason>

The operating system returned an error when "libinfo" tried to seek to the specified location (in hexadecimal) in the specified file. This message is followed by an interpretation of the error returned by the operating system.

"<file_name>" is not a library!

The file specified does not have the correct format for a library created with the "libgen" command.

** 'M' taken, others ignored ***

The 'm' and 'e' options are incompatible with the 'M' option. If the user specifies incompatible options, "libinfo" uses the 'M' option and ignores any others.

Unknown option '<char>' ignored.

An unknown option was found and ignored.

SEE ALSO

libgen
relinfo

link

Establish a new link to an existing file.

SYNTAX

link <file_name_1> <file_name_2>

DESCRIPTION

The "link" command establishes a new link to an existing file. If the command is successful, both <file_name_1> and <file_name_2> refer to the same file.

The user must have write permission in the parent directory in which the new link is created, and must have execute permission in the directory containing the original copy of the file. Only the system manager may make a link to a directory. A link cannot cross a volume boundary.

Arguments

<file_name_1> The name of the existing file to which to establish a link.

<file_name_2> The name to link to the existing file.

EXAMPLES

link /susan/.editconfigure .editconfigure

This example creates a file named ".editconfigure" in the user's working directory and links it to the existing file ".editconfigure" in the directory "/susan".

ERROR MESSAGES

Cannot link across devices

The specified file names reside on different volumes and, therefore, cannot be linked.

Entry already exists: <file_name_2>

The file specified by <file_name_2> must be a nonexistent file.

Entry does not exist: <file_name_1>

If the file to which the link is to be made does not exist, it is impossible to link the files.

Entry is a directory: <file_name_1>

The existing file specified is, in fact, a directory. Only the system manager can link to a directory.

Invalid option: <char>

The "link" command supports no options.

Path cannot be followed: <file_name>

One or more of the directories that make up the name of the file do not exist.

Permissions deny access: <file_name>

The user does not have permission to access the specified file. If the file is the existing file, <file_name_1>, the user does not have execute permission in the parent directory. If the file is <file_name_2>, the user does not have write permission in the parent directory.

Syntax: link <file_name_1> <file_name_2>

The "link" command expects exactly two arguments. This message indicates that the argument count is wrong.

SEE ALSO

copy
move

list

Write the contents of the specified file to standard output.

SYNTAX

list [<file_name_list>] [+l<num>]

DESCRIPTION

The "list" command writes the contents of the specified file to standard output. If the user specifies more than one file, the files are listed one after the other with no space between them.

The default file name is standard input. A plus sign, '+', may also be used as an argument to indicate standard input.

Arguments

<file_name_list>	A list of the names of the files to write to standard output. The default is standard input.

Options Available

l Include line numbers in the listing.

<num> The number of the line at which to begin listing the file.

EXAMPLES

1. list test
2. list test +120 >>test.out
3. list part_1 part_2 + part_3 >whole_thing

The first example writes the file "test" to standard output.

The second example also writes the file "test" to standard output. However, in this case standard output is redirected so that the listing is appended to the contents of the file "test.out". The listing is accompanied by line numbers and starts at line 20 of the file.

The third example writes the files "part_1" and "part_2", followed by the text entered from standard input, followed by "part_3", to the file "whole_thing".

ERROR MESSAGES

Error listing "<file_name>": <reason>

The operating system returned an error when "list" tried to write <file_name> to standard output. This message is followed by an interpretation of the error returned by the operating system.

Error opening "<file_name>": <reason>

The operating system returned an error when "list" tried to open the file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>": <reason>

The operating system returned an error when "list" tried to read the file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Invalid option: '<char>'. Command aborted!

The option specified by <char> is not a valid option to the "list" command.

Invalid starting line number. Command aborted!

The string used to specify the starting line of the listing either is not a string of digits or is too large.

load

The "load" command is the linking loader.

SYNTAX

load <file_name_list> [+aAbcCdDeFilLmMnNoPrsStTuU]

DESCRIPTION

The "load" command takes as input one or more relocatable binary modules and produces as output either a relocatable module or an executable module. The relocatable modules used as input should have been produced by the relocating assembler or the linking loader. Options are available for producing load and module maps as well as a global symbol table. Starting addresses for text and data segments can be adjusted for the particular hardware being used. The page size can also be adjusted. The loader can search libraries produced by the "libgen" utility in order to resolve external references.

The user can place all desired options in a file specified with the "load" command's 'F' option rather than specifying them individually on the command line. The operating system comes with one such file, the file "/lib/std_env", which describes the hardware environment. The loader always reads this file before processing any other options. It then processes options in the order in which they appear on the command line. If an option is specified more than once (e.g., once in a file and once on the command line), the last specification overrides all others.

Arguments

file_name_list> A list of files to load.

Options Available

a=<num> Specifies the minimum number of pages to allocate to this task at all times. The default is 0. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs.

A=<num> Specifies the maximum number of pages to allocate to this task at all times. The default is 0. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs.

b=<task_size> Specifies the size of the task, where <task_size> is "128K", "512K", "2048K", "8192K", "2M", or "8M". The default is "512K". If the argument specified by the user is not large enough, the "load" command adjusts it to the smallest possible size. The letters 'M' and 'K' can be either upper- or lowercase.

c=<module_type> Specifies the source code of the modules, where <module_type> is "ASSEMBLER", "C", "COBOL", "FORTRAN", or "PASCAL". The names can be specified in either upper- or lowercase.

C=<configuration> By default, the loader uses the configuration number of the current hardware. The user may, however, use the 'C' option to specify a configuration number which overrides the default. This option is useful when loading a module for a machine other than the one on which it is running.

d Sets the "no core dump" bit in the binary header.

D[=<hex_num>] Specifies the starting address of the data segment. If the user does not specify the option or specifies the option without an argument, the data segment immediately follows the text segment.

e Prints each occurrence of any unresolved external. By default, the loader prints only the first occurrence.

F[=<file_name>] Specifies the name of a file of options to process. The default file name is "ldr_opts". The 'F' option may be used repeatedly but may not be nested.

i Writes all global symbols to the symbol table of the binary file.

l=<library_name> Specifies the name of a library to search. The loader first searches the working directory, then the "lib" directory in the working directory, and finally the directory "/lib." Libraries are searched in the order specified on the command line. Up to five libraries may be specified in this manner. By default, unless the user specifies five libraries on the command line, the library "/lib/Syslib68k" is the last one searched.

L Does not search any libraries for unresolved externals.

m Produces load and module maps and writes them to standard output (see the 'M' option).

M=<file_name> Specifies the name of the file in which to put the output of the 'm' option (load and module maps) and the 's' option (a global symbol table). This information is purely textual. The user may edit or list the file like any other text file. If the 'm' or 's' option is used without the 'M' option, the loader sends the information to standard output.

n Produces an executable module with separate instruction and data space.

N=<module_name> Specifies the name to give to the file containing the module.

o=<file_name> Specifies the name to give to the binary file.

P=<hex_num> Specifies the page size. The hexadecimal number should always be a power of 2; otherwise, the results are unpredictable. The "load" command uses the page size to determine the starting address of the data segment when it immediately follows the text segment (the data segment starts at the next page boundary). The default is 0 (i.e., the loader rounds the starting address to the next even location after the end of the text segment).

r Produces a relocatable module as output. Do not search any libraries.

s Writes the global symbol table to standard output (see the 'M' option).

S=<hex_num> Specifies an initial stack size where the hexadecimal number is the number of bytes to reserve. The default is 0 (the system determines the size of the stack).

t Produces a shared-text executable module.

T=<hex_num> Specifies the starting address of the text segment. Default is 0.

u Does not print any unresolved messages when producing a relocatable module.

U=<trap_num> Sets the trap number for system calls. The default is hardware-dependent. The user can specify the argument as either "TRAP n" where 'n' is a number between 0 and 15 inclusive, or as a string of four hexadecimal digits which represent a bit pattern to use as an instruction instead of the system call.

EXAMPLES

```
1. load *.r +F=/lib/ldr_environ +t +l=Clib +o=tester
2. load t1.r t2.r +T=20000 +iN=mod +P=2000 +c=C +o=test
3. load sqrt +msM=loadmap +l=mathlib +i
4. load temp?.r +reo=combined.r
5. load t1.r t2.r +a=10 +A=100 +b=2M +l=testlib +do=test
```

The first example loads all files whose names end with ".r" in the working directory. The loader reads the file "/lib/ldr_environ" and processes the options therein. It uses the library "Clib" to resolve externals. The executable output module, which is a shared-text module, is named "tester".

The second example loads the the files specified and produces a binary file named "test". The internal module-name is "mod". The text segment begins at 20000 hexadecimal, and the data segment follows it at the next page boundary (page size 2000 hexadecimal). The source code is "C". All global symbols are inserted in the symbol table of the binary file.

The third example loads the file "sqrt" and produces an executable module named "sqrt.o". The loader searches the library "mathlib" for unresolved externals. It produces load and module maps, as well as a symbol table, and writes them to the file "loadmap". All global symbols are added to the symbol table of the binary file.

The fourth example loads the files in the working directory whose names match the pattern "temp?.r" and produces a relocatable module named "combined.r". The loader prints each occurrence of all unresolved externals rather than only the first occurrence of each. Because the 'r' option is specified, the loader does not search any libraries.

The fifth example loads the files "t1.r" and "t2.r" and produces the binary file named "test". The minimum page allocation is set to 10; the maximum, to 100. The task size of the module is set to 2 Megabytes. The executable module does not produce a core dump.

NOTES

- If the file "/lib/std_env" contains information about the starting address of the text segment, the data segment, or both, and if the user wishes to override this standard configuration, starting addresses for both text and data segments should be specified.

- If the user specifies page allocation values that don't make sense, the loader automatically adjust them according to the following rules:

  The value for the maximum is always greater than or equal to the value for the minimum. The value for the maximum can be 0, but if it is greater than 0, it must be at least 4.

SEE ALSO

Section 5, The Assembler and Linking Loader

login

Give a user access to the operating system.

SYNTAX

login <user_name>

DESCRIPTION

The "login" command gives a user access to the operating system. If the user does not have a password, the system automatically honors the command. If the user does have a password, the system requests it. If it is entered correctly correctly, the user is given access to the operating system. Otherwise, the system returns an error message, followed by a login prompt.

Arguments

<user_name>	The name of the user to put in contact with the operating system. If no <user_name> is supplied, the system prompts for it.

EXAMPLES

login leslie

This example tells the operating system to give the user whose user name is "leslie" access to the operating system.

NOTES

- The "login" command creates a file called ".home?" in the user's login directory. This file contains the full path name of the login directory, which is defined in the password file, "/etc/log/password". If the login program (also defined in the password file) is the shell program, it reads this file and deletes it. Thus, it knows what the user's home directory is. If the login program is not the shell program, the file ".home?" remains intact. This short file (it uses one block) does not affect the rest of the system.

ERROR MESSAGES

Login incorrect!

The combination of the user name specified and the password entered is invalid. This message is followed by a login prompt.

No "login" name specified.

The user did not specify a user name on the command line.

SEE ALSO

log
script
shell

move

Rename a file or move a file to another directory.

SYNTAX

```
move <file_name_1> <file_name_2> [+klps]
move <file_name_list> <dir_name> [+klps]
```

DESCRIPTION

The "move" command moves or renames one or more files. The first form of the command renames <file_name_1> to <file_name_2>. The second form moves each file named in <file_name_list> to <dir_name>. In either case, if there is already a file with the same name as the file created by the "move" command, it is overwritten without warning.

Directories and special files (block devices and character devices) may not be moved. The user must have write and execute permissions in the parent directory of each file being moved and in the directory to which the files are moved. Each original file is removed.

A file may not be moved from one device to another unless the user has read permission on the file. A file may not be moved to itself.

Normally the "move" command links the new file to the original file and deletes the original one. Thus, a link between files on different devices is not permitted; if you attempt to "move" a file to a different device, the original file is copied to the new file, then the original file is deleted.

Arguments

<file_name_1> The name of the file to move or rename.

<file_name_2> The name of the file to which to move <file_name_1>.

<dir_name> The name of the directory to which to move all the specified files.

Options Available

k Do not delete the original file.

l List the name of each file as it is moved.

p Prompt for permission to replace existing files.

s Stop as soon as an error is encountered.

EXAMPLES

1. move test oldtest +l
2. move test /elaine
3. move test /elaine/oldtest +kp
4. move * /elaine +s

The first example renames the file "test" in the working directory; the new name is "oldtest." The "move" command issues a message describing the move.

The second example moves the file "test" from the working directory to the directory "/elaine". The last component of the file name is preserved, so the name of the new file is "/elaine/test".

The third example moves the file "test" from the working directory to the directory "/elaine" and renames it "oldtest." If the file "/elaine/oldtest" already exists, the user is prompted for permission to delete the file. If permission is denied, the move does not take place. Even if the move takes place, the original files remain intact.

The fourth example moves all the files in the working directory to the directory "/elaine". The last component of each file name is preserved. The command aborts if it encounters an error.

MESSAGES

<file_name_1>" copied to "<file_name_2>"

This message is produced only if both the 'l' and 'k' options are specified. It means that <file_name_1> has been copied to <file_name_2>, but that the original file remains intact. This message indicates that the two files are on different devices.

"<file_name_1>" linked to "<file_name_2>"

This message is produced only if both the 'l' and 'k' options are specified. It means that the two files have been linked but that the original file remains intact (the user specified the 'k' option).

"<file_name_1>" moved to "<file_name_2>"

This is the normal message issued by the "move" command. It means that <file_name_1> has been either linked or copied to <file_name_2>, and that <file_name_1> has been deleted.

ERROR MESSAGES

Cannot move a block special file: <file_name>

The file <file_name> is a block special file (block device) and may not be moved.

Cannot move a character special file: <file_name>

The file <file_name> is a character special file (character device) and may not be moved.

Cannot move across devices: <file_name>

The file <file_name> is read-protected and, therefore, cannot be moved across devices.

Directory is not accessible: <dir_name>

The user does not have the necessary permissions (write and execute) to move a file to <dir_name>.

"<file_name_1>" and "<file_name_2>" are the same file.

The user tried to move a file to itself, which if allowed would destroy the file. If <file_name_1> and <file_name_2> are different, they are links to the same file.

Permissions deny access: <file_name>

The user does not have write permission in the parent of the specified directory.

SEE ALSO

copy
link

owner

Change the owner of a file.

SYNTAX

owner <new_owner> <file_name_list>

DESCRIPTION

The "owner" command changes the owner of the specified file. Only the system manager may execute this command.

Arguments

<new_owner> The user name or user ID of the new owner of the file.

<file_name_list> A list of the names of the files for which to change the owner.

EXAMPLES

1. owner system /john/*

2. owner 110 /john/*

The first example changes the owner of all the files in the directory "/john" to "system".

The second example changes the owner of all the files in the directory "/john" to the user whose ID is 110.

ERROR MESSAGES

Error changing owner for "<file_name>": <reason>

The operating system returned an error when "owner" tried change the owner of the specified file. This message is followed by an interpretation of the error returned by the operating system.

"<name>" is not a valid user name.

The specified name is not in the password file and, therefore, is not a valid user name.

<num> is not a valid user identification number.

The specified number is not in the password file and, therefore, is not a valid user ID.

Syntax: owner <new_owner> <file_name_list>

The "owner" command expects at least two arguments. This message indicates that the argument count is wrong.

You must be system manager to run "owner".

Only the system manager may execute the "owner" command.

password

Set or change a user's password.

SYNTAX

password [<user_name>]

DESCRIPTION

The "password" command sets or changes a user's password. Only the system manager may change another user's password. When a user other than the system manager invokes the command, the operating system prompts for the existing password (if there is one). If the password is entered correctly, the system prompts for the new password. Generally, a password should contain between five and eight random characters. After the new password is entered, the system prompts for it again to verify it. If the second entry agrees with the first, the password is entered in the password file. In order to maintain the secrecy of the password, the operating system does not echo the characters typed in response to the prompts for either the existing or the new password.

To remove a password, enter a carriage return for the new password.

Arguments

<user_name>	The name of user whose password is being changed. The default is the user invoking the command.

EXAMPLES

1. password
2. password greg

The first example changes the password of the user who invoked the command.

The second example uses the command form that can be used only by the system manager. It changes the password associated with the user name "greg".

ERROR MESSAGES

Cannot find "<user_name>" in the password file.

The file "/etc/log/password" does not contain an entry for the user <user_name>.

Cannot find your name in the password file.

The file "/etc/log/password" does not contain an entry for the user issuing the command. This situation is extremely unlikely to occur.

Error linking "/tmp/pswd" to "/etc/log/password": <reason>

The operating system returned an error when "password" tried to link the new version of the password file to the old password file. This message is followed by an interpretation of the error returned by the operating system.

Error opening "<file_name>": <reason>

The operating system returned an error when "password" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error unlinking "<file_name>": <reason>

The operating system returned an error when "password" tried to unlink the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error writing "<file_name>": <reason>

The operating system returned an error when "password" tried to write to the specified file. This message is followed by an interpretation of the error returned by the operating system.

Only the system manager may change another's password.

Use of the form of the "password" command that takes an argument is limited to the system manager.

Password not correct. Permission denied!

The user did not enter the existing password correctly.

Retry different password unchanged.

The first and second entries of the new password were not identical. The password command aborts, leaving the original password in place.

Syntax: password [<user_name>]

The "password" command expects no more than one argument. This message indicates that the argument count is wrong.

System busy - try again later.

The file "/tmp/pswd", which must be created by the "password" command already exists. Either someone else is using the command or it was interrupted before it had a chance to delete the temporary file. If no one is using the command, you should login as "system" and delete the file "/tmp/pswd".

path

Write the path name of the working directory to standard output.

SYNTAX

path

DESCRIPTION

The "path" command writes the path name of the working directory, followed by a carriage return, to standard output. The path name, also called the file specification, is the unique path from the root directory through the directory tree to the file in question.

EXAMPLES

path

This example is the only valid form of the "path" command. It writes the name of the working directory, followed by a carriage return, to standard output, which defaults to the user's terminal. Of course, the user may redirect standard output.

ERROR MESSAGES

Directory structure is corrupt

The directory path from the root directory, '/', to the working directory is corrupt. Therefore, the "path" command cannot determine the path name of the working directory.

SEE ALSO

chd

perms

Change the permissions associated with a file.

SYNTAX

perms <perms_list> <file_name_list>

DESCRIPTION

Every time a user creates a file, the operating system assigns it a set of permission bits which determines whether or not the file's owner and other users may read, write, or execute the file. The permissions assigned depend on the command used to create the file. The editor, for example, creates all files with "rw-rw-" permissions, which allow the user who owns the file, as well as other users, to read and write, but not execute, the file. The default permission for "crdir" are "rwxrwx"; for create, "rw-rw-"; for "makdev", "rw-r--".

Read permission allows a regular file to be read. A user cannot execute commands such as "list" and "copy" without read permission on the file in question. Write permission allows a file to be modified. Execute permission allows the name of the file to be used as a command.

Permissions for directories are similar to those for normal files. Read permission allows the user to read file names that are actually in the directory. Write permission allows the user to create and delete files in the directory. Execute permission allows the directory to be searched for a name used as part of a file specification or file name. The user must have execute permission to successfully use a directory as the argument to the "chd" command.

In addition to these permissions, each file has associated with it a user ID bit. If this bit is set for a given file, any user executing the file has the same privileges as the file's owner for the duration of the task.

The "perms" command changes the permission bits associated with a file. Only the owner of a file or the system manager may change the permissions associated with it.

Arguments

<perms_list> The list of permission bits to alter. Permission bits not mentioned are not changed.

<file_name_list> A list of the names of the files for which to alter the permissions.

Format for Arguments

<perms_list> The first character of an element in the permissions list specifies whether the argument applies to the user who owns the file ('u') or to others ('o'). The second character specifies whether to add ('+') or remove ('-') the permissions in question. The second character is followed by one, two, or three of the characters 'r', 'w', and 'x' (for read, write, and execute). The user ID bit is set or cleared with one of the following arguments: "s+" or "s-".

EXAMPLES

1. perms o-wx inventory
2. perms o+x u+x script
3. perms o-rw o+x s+ inventory script

The first example removes write and execute permissions for other users from the file "inventory" in the working directory.

The second example gives execute permissions on the file "script" to both the user who owns it and to other users.

The third example removes read and write permissions for others from the files "inventory" and "script". It also sets execute permissions for others, as well as the user ID bit. Thus, although other users may neither read from nor write to the files, they may execute them. While they are executing them, they have the same permissions on all files as the owner of these files does.

ERROR MESSAGES

Error changing permissions for "<file_name>": <reason>

The operating system returned an error when "perms" tried change the permissions on the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error processing "<file_name>": <reason>

The operating system returned an error when "perms" tried to determine the original permissions on the file. This message is followed by an interpretation of the error returned by the operating system.

Syntax: perms <perms_list> <file_name_list>

The "perms" command expects at least two arguments. This message indicates that the argument count is wrong.

Unrecognizable character, '<char>', found in permissions list. Command aborted!

A character following a plus or minus sign in an element in the permissions list was not an 'r', 'w', or 'x'. The command aborts without altering any permissions.

SEE ALSO

dir
dperm

relinfo

Display information about an object file.

SYNTAX

relinfo <file_name_list> [+ehrs]

DESCRIPTION

The "relinfo" command examines an object file or all the modules in a library and displays information about the binary header, the symbol table, and both the relocation and external records. Normally, "relinfo" displays all the information. The available options restrict the display to the specified information.

Arguments

<file_name_list> A list of the names of files to report on.

Options Available

e Display only information about external records.

h Display only information about the binary header.

r Display only information about relocation records.

s Display only information about the global symbol table.

EXAMPLES

1. relinfo tester
2. relinfo /lib/mathlib +h
3. relinfo reporter +se

The first example displays information about the binary header, the symbol table, and both the relocation and external records in the object file "tester" in the working directory.

The second example displays the information about the binary headers from all the modules in the library "/lib/mathlib".

The third example displays the information about both the relocation and external records in the file "reporter" in the working directory.

ERROR MESSAGES

Error opening "<file_name>" : <reason>

The operating system returned an error when "relinfo" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error reading "<file_name>" : <reason>

The operating system returned an error when "relinfo" tried to read the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error seeking to <location> in "<file_name>" : <reason>

The operating system returned an error when "relinfo" tried to seek to the specified location (in hexadecimal) in the specified file. This message is followed by an interpretation of the error returned by the operating system.

file_name>" is not a binary file!

The specified file does not have a valid binary header.

Unknown option '<char>' ignored.

An unknown option was found and ignored.

SEE ALSO

libgen
libinfo
load
asm

remote

Communicate with a host computer via the RS-232 port, "/dev/comm."

SYNTAX

remote [+l= filename [+n]]

DESCRIPTION

The utility "remote" allows the 4404 to be used as a terminal to a remote host computer connected to the "/dev/comm" port.

"Remote" allows you to capture both sides of a session with a host into a disk file for later editing and review. In addition, this utility also allows file transfers to and from the host under control of a host program.

Options Available

+l= filename

Output from the host will be directed to the specified file in addition to being sent to the terminal emulator and appearing on the screen. This function can be toggles on and off using function key F3. +n This options specifies that linefeed characters be ignored when directing to the file specified by the +l option. The +l option must be specified for this option to have any meaning.

FUNCTION KEY ACTIONS

F1 Terminates remote.

F2 Create and enter a subshell. Any executing file transfers will continue uninterrupted.

F3 Toggles output to file specified by the +l option on and off.

FILE TRANSFERS

Remote supports a file transfer protocol which works in conjunction with a program running on the remote host. The 'C' source code for a sample of such a program, which will run under the Unix<tm> operating system, may be found in "/samples/xfer.c"

CONFIGURING THE COMMUNICATIONS PORT

The "commset" command is used to set the various parameters of the communications port. For example, the baud rate of the part may be set with a command like:

```
commset baud=9600
```

See the documentation on the "commset" command for further information on configuring the communications port.

remove

Remove the specified file from the system.

SYNTAX

remove <file_name_list> [+dklpw]

DESCRIPTION

The "remove" command removes the specified file, which may be any type of file, from the file system. The user must own the file, must have write permission in the parent directory of the file being removed and, by default, must also have write permission in the file itself. Restrictions on deleting a directory are discussed with the options.

Arguments

<file_name_list>	A list of the names of files to remove from the file system. The list may include regular files, special files, and directories.

Options Available

d — If the specified file is a directory and it is empty, delete it. By default, the "remove" command does not delete directories.

k — If the specified file is a directory, delete it and all the files it contains.

l — List the name of each file as it is removed.

p — Prompt for permission to remove each file. The file is removed if the user responds to the prompt with a 'y'.

w — Prompt for permission to remove files for which the user does not have write permission. By default, the "remove" command does not delete such files. The file is removed if the user responds to the prompt with a 'y'.

q — Quiet mode. Do not issue messages.

EXAMPLES

1. remove first_file dir_file second_file +w
2. remove first_file dir_file second_file +dp
3. remove first_file dir_file +kl

The first example removes the files "first_file" and "second_file", prompting for permission to do so if the user does not have write permissions in the file. The file "dir_file" is not removed because it is a directory.

The second example prompts for permission to remove "first_file" and "second_file" (assuming the user has the proper permissions). It also prompts for permission to remove "dir_file" if the directory is empty.

The third example removes "first_file" and "dir_file" from the file system. In addition, it descends the directory structure of "dir_file", deleting the directory itself as well as every file. The command lists the name of each file as it is deleted.

CAUTION

The "remove" command, especially when executed with the 'k' option, is an extremely powerful and potentially destructive command.

ERROR MESSAGES

Cannot delete the root directory: "/"

The user tried to delete the root directory.

Directory "<dir_name>" is not empty.

The "remove" command cannot delete a nonempty directory unless the user specifies the 'k' option.

Error deleting "<file_name>": <reason>

The operating system returned an error when "remove" tried to delete <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error deleting "." in "<dir_name>": <reason>

The operating system returned an error when "remove" tried to delete the "." entry in <dir_name>. This message is followed by an interpretation of the error returned by the operating system.

Error getting status for "<file_name>": <reason>

The operating system returned an error when "remove" tried to read the fdn for <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error removing "<file_name>": <reason>

The operating system returned an error when "remove" tried to remove <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Invalid option: '<char>'

The option specified by <char> is not a valid option to the "remove" command.

Syntax: remove <file_name_list> [+dklpw]

The "remove" command expects at least one argument. This message indicates that the argument is wrong.

You do not own "<file_name>".

The user may not delete a file that is owned by someone else.

SEE ALSO

deluser

rename

Change the name of the specified file.

SYNTAX

rename <file_name_1> <file_name_2>

DESCRIPTION

The "rename" command changes the name of the specified file. If a file named <file_name_2> already exists, it is deleted without warning.

ARGUMENTS

<file_name_1> The name of an existing file.

<file_name_2> The new name for <file_name_1>.

EXAMPLES

1. rename test oldtest
2. rename test /elaine/oldtest

The first example changes the name of the file "test" in the working directory to "oldtest". If a file named "oldtest" already exists, it is deleted without warning.

The second example changes the name of the file "test" in the working directory to "/elaine/oldtest".

ERROR MESSAGES

Error renaming "<file_name_1>": <reason>

The operating system returned an error when "rename" tried change the name of <file_name_1>. This message is followed by an interpretation of the error returned by the operating system.

Error renaming to "<file_name_2>": <reason>

The operating system returned an error when "rename" tried to assign the new file name. This message is followed by an interpretation of the error returned by the operating system.

Error unlinking "<file_name_1>": <reason>

The operating system returned an error when "rename" tried to unlink <file_name_1> from the new file. This message is followed by an interpretation of the error returned by the operating system.

File "<file_name_1>" does not exist!

The first name on the command line must be the name of an existing file.

File "<file_name>" is a directory!

The "rename" command can neither rename a directory nor assign a directory name to an existing file.

Syntax: rename <file_name_1> <file_name_2>

The "rename" command expects exactly two arguments. This message indicates that the argument count is wrong.

SEE ALSO

move

restore

Catalog or Copy files from the floppy device back onto the file system.

SYNTAX

restore [+ bBCdLlnp] [+ a =days] [file ...]

DESCRIPTION

The "restore" command is used to copy backup files from the floppy device back onto the file system. Although the program is named Restore, it can operate in two distinct modes, selected by options: catalog mode and restore mode. Catalog mode lists the contents of the backup device in much the same format as that used by the "dir" and "ls" commands. Restore mode retrieves files or directories from a backup device.

The "restore" command retrieves backup files and directories from /dev/floppy only. You should not attempt to "mount" a backup diskette; the only way to read disks written by "backup" is to use the "restore" command. The only other command that you should use on a backup diskette is "devcheck".

Arguments

<file_name_list>	List of the names of files and directories to process. Default is the working directory.

If you specify a directory name as an argument in restore mode, the program processes only the files within that directory. If you also specify the 'd' option, the program restores all files within the given directory and its subdirectories.

Options Available

a=<days>	Restore only those files that are less than the specified number of days. A value of 0 specifies files created since midnight on the current day; a value of 1 specifies files created since midnight of the previous day, and so forth.
b	Print sizes of files in bytes.
B	Do not restore files that end in ".bak".

C	Print a catalog of the files on an existing backup. If you specify the 'C' option, "backup" ignores all the names in <file_name_list>.
d	Restore entire directory structures.
l	List file names as they are restored.
L	Do not unlink files before restoring.
n	Only restore a file if the copy on the back up device is newer than the copy at the destination. If the destination file does not exist, the program restores the file (unless prohibited by another option, such as the 'B' option). The 'n' option may be used only in restore mode.
p	Prompt you with each file name to determine whether or not the restore procedure should be performed on that particular file.

"restore" normally works in a quiet mode. The 'l' option allows you to see what the program is actually doing.

EXAMPLES

1. restore +lR

2. restore +lRn file1 dir2

3. restore +C >catalog

The first example restores all of the files, excluding subdirectories and their contents, from the backup diskettes you are prompted to insert in the flexible disk drive.

The second example restores the file "file1" from the backup. It then restores the files contained in "dir2" on the backup, creating the directory "dir2" if necessary. This example does not restore any subdirectories in "dir2" or any files or directories contained in subdirectories in "dir2".

The third example catalogs the files on the backup set and stores it in a file called "catalog."

NOTES

- In restore mode, file names or directory names on the command line are used to select the files or directories to be restored. The program searches the entire backup for each argument specified. If multiple files satisfy the restoration criteria, the program restores them all, destroying the older version as the new one is restored. Thus, to ensure proper restoration, you must provide all backup volumes (in order) for each argument.

- When files are restored, they are generally restored to the same directory location as you specified when they were backed up. As files are backed up, "backup" makes an indication of the path name for each file. When files are restored, "restore" uses the path name to place the file in its proper directory location. If the path name is relative (i.e., does not begin with '/'), the path name of the restored directory is also relative. Thus, files backed up with a relative path name may be restored to a directory location different from the one in which they were created.

  An example should make this clear. If the working directory is backed up, either by specifying no source files or by using the directory name '.', the files are backed up with a relative path of '.'. When these files are restored, they are placed in the directory '.', which might not be the same directory they originally came from. This feature allows the manipulation of entire file systems in a general fashion. To specify a unique directory location for a file, you should specify its entire path name, starting with '/'.

- It is possible to restore backed up data onto the device currently being used as the root device or system disk. Two possible problems arise, however. First of all, if the operating system is restored from a backup, the result is not bootable. In such a case, the file must be copied from the original master diskette and installed in order to allow booting. The second problem occurs if the shell program or the device "tty00" is restored over the current shell or "tty00". This operation leaves unreferenced files in the file system. Unreferenced files must be corrected with the "diskrepair" command. In general, it is always a good idea to run "diskrepair" on the root device after restoring backed-up data to it.

MESSAGES

Several of the following messages prompt you for a positive or negative response. The program interprets any response that does not begin with an upper or lowercase 'n' as a positive response.

```
Catalog of backup on "<file_name>"
Restore backup from "<file_name>"
```

These messages are printed when "backup" begins. They notify you of the function about to be performed.

```
Restore "<file_name>" (y/n)?
```

If you specify the 'p' option, the program prints one of these prompts before it takes any action. A response of 'n' or 'N' indicates that the operation should not be performed for the given file. Any other response is interpreted as "yes".

```
Insert next volume - Hit C/R to continue:
```

This prompt is issued when the program needs a new backup volume. You should type a carriage return only when the next volume has been placed in the device.

```
link "<file_name_1>" to "<file_name_2>"
copy "<file_name>"
Copying from "<dir_name>"
```

The program prints these messages as it takes the corresponding action during a creation operation.

```
This is Volume #<number_1> -- Expected Volume #<number_2> --
Continue?
```

The program expects you to insert volumes in sequential order. If a volume appears out of order, "backup" prints this message. If you type anything except an 'n' or an 'N' as the first character of the response to the message, "backup" ignores the fact that the volumes are out of order and continues with the backup. Otherwise, it prompts you for another volume. It is important to insert volumes sequentially because "backup" cannot correctly restore files that are broken across volumes if the volumes are inserted out of order.

Volume <number> of "<vol_name>"

Whenever a new volume is inserted and properly validated, the program prints this message, which indicates the name of the backup volume and its sequence number.

ERROR MESSAGES

dev_name>" is not a block device

The destination device for the backup must be a block device. This message indicates that the specified device (that is always the first argument) is not such a device.

file_name>" not located - try again?

When using the program in restore mode, you may specify which files or directories to restore. If the program cannot find a specified file or directory after searching the entire backup, it prints this message. If the response is not 'n' or 'N', the program searches the entire archive again. This option is allowed because volumes need not be inserted in order of their creation when the program is in restore mode. If one volume is left out or if the final volume is inserted before the entire archive has been processed, some files might not be processed. Note that if you specify more than one file name or directory name, the program processes the entire archive for each file before proceeding to the next one.

Formatting not allowed during Catalog/Restore

You may not format a disk if the program is in either catalog or restore mode.

Read error! - file "<file_name>"

An I/O error occurred during the transfer of a file either to or from the backup. An auxiliary message is printed indicating the nature of the error. The program tries to continue for all errors except "device full" during restore mode.

Unknown option: <char>

The option specified by <char> is not a valid option to the "backup" command.

** Warning: directory "<dir_name>" is too large!
** Some directories were ignored
**Warning: directory "<dir_name>" is too large!
** Some files were ignored

The program uses some internal tables during the back up process (not during restore or catalog). If the limits of these tables are exceeded (highly unlikely), these messages are printed.

SEE ALSO

backup

script

The script execution shell.

DESCRIPTION

The program named"script" is a command interpreter used primarily to execute commands from a file. It can be run as an interactive interface, but does not support aliases, history, and environmental variables that are available under "shell."

If you run "script" as an interactive shell, it collects and interprets your commands and executes some built-in commands ("chd", "dperm", "jobs", "log", "login", "time", and "wait") itself. It passes others to the operating system kernel which, in turn, performs the operations requested.

A "script" command line consists of a command name, which may be followed by arguments, options, or both. All elements of the command line must be separated by either spaces or commas. The command may be one of the commands supplied with the operating system, the name of a binary file produced by either the assembler or a compiler, or the name of a text file (with execute permission turned on) which contains a series of commands to execute. In all cases the script program spawns a child-task which executes the specified command or commands.

Search Path

The list of directories searched by the script program is known as the search path. Because most commands reside on disk, the script program must locate the command before executing it. By default, the script program sequentially searches the following directories: your working directory, "<home_dir>/bin", and "/bin". If you are the system manager, the system also searches the directory "/etc" immediately after searching "<home_dir>/bin". (The home directory is your login directory, as specified in the password file.)

Background Tasks

If you follow a command with an ampersand, '&', the script program, as usual, spawns a child-task which executes the command. However, in this case the script does not wait for the task to complete. Thus, you may start another command while the first one is executing. A single script program can support a maximum of five of these "background tasks". Each time you send a task to the background, the script program reports the task ID assigned to that task, preceding it with a 'T', which is not part of the task ID. The user may need the task ID to execute the "wait" or "int" command. The task ID may also be obtained by executing the "jobs" command, which returns the task ID and starting time of all background tasks originated by you at the current terminal from the script program. The ampersand may be used following a single command or separating one task from another on the command line.

Multiple Commands on a Line

You may specify more than one command on a command line by separating them with any of several special symbols.

The script program sequentially executes commands that are separated by a semicolon, ';'. If a task terminates abnormally, the script program stops executing the command line.

Two additional command separators, the conjunction operator ("&&") and the disjunction operator ("||"), are available. With these separators, execution of the command following the operator is dependent on the outcome of the command preceding it. A command is "true" if it terminates with a termination status of zero, indicating successful completion, and "false" if it terminates with a nonzero termination status, indicating failure. When two commands are separated by the conjunction operator, the script program executes the second one only if it completes the first one successfully (it is "true"). When two commands are separated by the disjunction operator, the script program executes the second one only if the first one fails (it is "false").

Normally, the command line is evaluated from left to right; however, parentheses may be used to group commands. Commands in parentheses are treated as a single command. Commands separated by a pipe (see Redirected I/O) are also treated as one command.

The processing of the command separators may be summarized as follows:

&& If the command preceding the conjunction operator succeeds, the script program tries to execute the next command. If the command preceding the conjunction operator fails, the script program looks for a disjunction operator. If it finds one, it tries to execute the command which follows it. If it does not find one, processing of the command line ceases.

|| If the command preceding the disjunction operator succeeds, the script program looks for a semicolon, ';'. If it finds one, it tries to execute the command which follows it. If it does not find one, processing of the command line ceases. If the command preceding the disjunction operator fails, the script program tries to execute the next command.

; If the command preceding a semicolon succeeds, the script program tries to execute the next command. If the command preceding a semicolon fails, processing of the command line ceases.

& Whether the command preceding a single ampersand succeeds or fails, the script program processes the next command on the command line.

Consider the following example:

<task_1> && <task_2> || <task_3> && <task_4>

The script program first tries to execute <task_1>. If the task is unsuccessful, the script skips <task_2> and proceeds to <task_3>. If <task_3> fails, the script program skips <task_4>; if <task_3> succeeds, it tries to execute <task_4>. If, however, <task_1> succeeds, the script program tries to execute <task_2>. If <task_2> also succeeds, the script program skips the rest of the command line. If, after the successful execution of <task_1>, <task_2> fails, the script tries to execute <task_3>. If and only if <task_3> succeeds, it goes on to <task_4>.

The use of parentheses can change the interpretation of the same set of commands separated by the same operators:

<task_1> && (<task_2> || <task_3>) && <task_4>

In this case, the script once again begins by trying to execute <task_1>. If it fails, the script program skips the remaining tasks. If, on the other hand, <task_1> is successful, the script program spawns a subshell (because of the presence of the parentheses). This subshell tries to execute <task_2> and, if and only if it fails, it tries to execute <task_3>. If <task_2> succeeds, it returns a termination status of "true" to its parent script. If <task_2> fails but <task_3> succeeds, it also returns a termination status of "true". If, however, both <task_2> and <task_3> fail, the termination status returned is "false". If the termination status returned by the subshell is "true", the parent script tries to execute <task_4>.

Termination Status

Normally, the script program does not report the termination status of a command it executes unless the task terminates abnormally (because of a program interrupt). A list of the possible program interrupts appears in the documentation of the "int" command. The script program does, however, always report the termination status of a background task, even if it terminates normally.

Redirected I/O

The script program associates three files with every command it executes: standard input, standard output, and standard error. Standard input is the file from which a command takes its input. Standard output is the file to which a command sends its output. Standard error is the file to which many error messages are directed. By default, the system uses your keyboard as standard input and your terminal as both standard output and standard error. However, you can direct the script to use another file for any of these standard files. This process is known as I/O redirection.

The symbol '<' tells the script program to take its standard input from the file whose name follows the symbol. Similarly, the symbols '>' and '' are used to send standard output and standard error to a file. The file to which standard input is redirected must already exist. However, if the file to which standard output or standard error is redirected does not exist, the system creates it. In fact, if the file does already exist, the system deletes the contents of the file before executing the command. To avoid this effect, you may use the ">>" symbol to direct the script program to append data to the file specified as standard error or standard output. For example, you might add the results of the "compare" command to one of the pre-existing files.

It is also possible to redirect standard output or standard error (or both) to another task. This form of redirection is accomplished by using a "pipe". A pipe is a function that connects programs so that the output from one program becomes the input for another. Standard output is piped from one task to another by using one of the symbols '¦' or '^'. For instance, the following example lists all the files in the working directory, formats the listing with the "page" command, and prints the listing on the printer "/dev/printer."

```
ls . ¦ page ¦ /dev/printer
```

Similarly, you can redirect standard error with either of the symbols "¦" or "^".

Although you can place many pipes on the command line, a single task can support only one pipe. Thus, you cannot pipe standard error and standard output to separate tasks. It is possible, however, to duplicate standard error onto standard output and to redirect them both to the same task. You have a choice of symbols for duplicating standard error onto standard output: ">%" or "%>". Neither of these symbols takes an argument. After duplicating standard error onto standard output, you redirect standard output to a file or a task in the usual way. Whenever standard error and standard output are routed to the same destination, their contents may be intermingled. For instance, you can get a listing of all the files in the working directory, redirect both standard error and standard output to the "page" command, and print the results on the printer "/dev/printer" with the following command:

```
ls . > ¦ page ¦ /dev/printer
```

Finally, the following constructions redirect I/O from or to the null device, "/dev/null": "<-" for standard input, ">-" for standard output, and "-" for standard error. If either standard output or standard error is redirected to the null device, its contents are lost. If the null device is used as standard input, an end-of-file character is read.

Continuation of the Command Line

Command lines may be continued across more than one physical line by terminating each line, except the last, with a backslash character, " ," immediately followed by a carriage return. As an interactive shell, "script" uses the prompt "+>" to indicate that the line being entered is a continuation of the previous line. When the script program processes the line, it replaces the backslash and the carriage return with a space. Typing a line-delete character (control-U) only affects the physical line being typed. You may delete previous lines of a continued command line by typing a keyboard interrupt (control-C), which deletes the entire command line.

Pattern Matching Characters

The operating system recognizes several characters, known as pattern matching characters, which allow you to specify files with similar names without typing each name individually. The special characters are the asterisk, '*'; the question mark, The script program matches these special characters to characters in the filenames in the specified directory. If the matching character appears in the last component of the file name, the script tries to match it to the names of all files in the specified directory (by default, the working directory). If the matching character appears in any other position in the file name, the script tries to match it to the names of directories only.

An asterisk in a command line matches any character or characters, including the null string but not including a leading period. Thus, the command

```
list *.bak
```

lists all files in the working directory whose names end in ".bak" and do not begin with a period.

The question mark matches any single character except the null character or a leading period. For example, the command

```
list chapter_?
```

lists all files whose names begin with the string "chapter_" and end with a single character other than the null character.

You can use more than one matching character at a time. For instance, the command

```
list *.?
```

lists all files in the working directory whose names end with a period followed by a single character (except those whose names begin with a period).

Square brackets allow you to specify a set of characters to use in the matching process. The set of characters is defined by listing individual characters or by specifying two characters separated by a hyphen. In the former case, the script program looks for all file names which use any one of the enclosed characters in the appropriate place. In the latter, the two characters specify a class of characters containing the two characters themselves and any characters which lexically fall between them in the ASCII character set. For example, if your working directory contains nine files named "chapter1", "chapter2", "chapter3", and so forth, the following command lists the first three chapters, the fifth chapter, and the last three chapters:

```
list chapter[1-357-9]
```

If the script program cannot find a match for any of the arguments containing matching characters, it aborts the command. If it finds a match for at least one argument containing matching characters, it ignores any other arguments containing matching characters for which it cannot find a match.

If a filename actually contains one of the matching characters or either a space or a comma, you must enclose the name in single or double quotation marks. In such a case the script program passes the arguments to the command without performing any character matching.

"script" Scripts

A "script" script is a file that contains a list of commands for the script program. Such a file might consist of a list of commands that are frequently executed in sequence, or of a single, lengthy command that is often used. If you set execute permissions on such a file, the name of the file can be used as a command.

You may add to the versatility of a "script" script by using arguments within the script. The arguments are specified within the script as "$1", "$2", "$3", and so forth. The argument "$0" specifies the name of the calling program. These arguments may appear anywhere in a command argument.

If an argument being passed to a command actually contains an ampersand, the argument must be enclosed in single quotation marks so that the script program does not try to perform any substitution. Note that single quotation marks prevent both substitution of arguments and the expansion of matching characters, whereas double quotation marks prevent the expansion of matching characters but allow the substitution of arguments.

The script program supports several commands that are used exclusively with "script" scripts. These commands--"verbose", "exit", "proceed", and "sabort"--are discussed in the following paragraphs.

"verbose"

When the script program executes a script file, it does not normally echo the commands being executed. The "verbose" command causes the script program to echo commands from a script file as they are executed. Each line that is echoed is preceded by two hyphens and a space character.

The "verbose" command may be called without arguments or with one argument, which must be one of the strings "on" or "off". If called without an argument, the default is "on". The command may be executed by the login script or may be part of a script script. The verbose attribute is always passed from a parent script program to a child shell, but never from a child to a parent.

"exit" and "proceed"

"script" permits a limited amount of control over the processing of script files. "shell" sequentially processes commands in a script file until one of the commands fails or it reaches the end of the file. If a command fails, "script" begins to search the remainder of the script file for a line that contains one of the commands "exit" or "proceed". If it encounters one of these commands, "script" resumes processing the script after that command. The only difference between "exit" and "proceed" commands is that during successful execution of a script file "script" stops processing the file if it encounters an "exit" command, whereas it ignores a "proceed" command. The search for both these commands takes place before both the substitution of any arguments and the expansion of any matching characters. Thus, the script program does not see an "exit" or "proceed" command that is created as the result of either of these processes.

Here's an example of the "proceed" command:

```
/etc/mount /dev/floppy /usr2
/usr2 runjob
echo "Successful execution."
proceed
/etc/unmount /dev/floppy
```

In this example, "script" mounts a disk and tries to execute the command "/usr2/runjob" on that disk. If the command succeeds, "script" echoes the message, "Successful execution." and proceeds to unmount the disk. If the command fails, "script" skips all commands between the one that failed and the "proceed" command. It resumes execution with the "unmount" command. Thus, if "/usr2/runjob" fails, your disk is unmounted, but no message is sent to standard output.

By adding an "exit" command you can modify this example to notify you of either successful or unsuccessful execution:

```
/etc/mount /dev/floppy /usr2
/usr2/runjob
/etc/unmount /dev/floppy
echo "Successful execution."
exit
/etc/unmount /dev/floppy
echo "Unsuccessful execution."
```

Here, if "/usr2/runjob" succeeds, the script program continues execution with the "unmount" command and echoes the string "Successful execution." to standard output. The "exit" command then causes the script program to stop processing the script because it encounters the "exit" command during normal execution. If "/usr2/runjob" fails, the script program skips all commands until it encounters the "exit" command. It then resumes execution with the "unmount" command and echoes the string "Unsuccessful execution." to standard output.

"sabort"

The "sabort" command can be used to turn off the search for either an "exit" or "proceed" command, thus forcing execution of every command in the script, regardless of the failure of a command.

"sabort" may be called without arguments or with one argument, which must be one of the strings "on" or "off". When "sabort" is "on", "script" looks for an "exit" or "proceed" command whenever a command in the script fails. When "sabort" is off, "script" processes every command in the script. If you execute the "sabort" command without an argument, it both rescinds the effect of any previous "sabort on" and fails. Thus, if "script" is executing a script, "script" immediately begins looking for an "exit" or "proceed" command.

The "sabort" command may be executed by a login shell (if you use "script" as your shell) or may be part of a "script" script. The attribute is always passed from a parent program to a child shell, but never from a child to a parent.

The system also supports startup files for individual users. Whenever a user logs in using "script" as an interactive shell, the script program looks for a file named ".startup" in your home directory (as defined in the password file). If the file exists and you have read permissions in it, "script" executes the file before issuing the system prompt.

The script program can also be used as a command in its own right. This form is used primarily to execute a "script" scriptfile for which execute permissions are not set, to call the script program from another program, or in the password file.

SYNTAX

script [abclnvx] [<argument_list>]

DESCRIPTION OF THE "SCRIPT" COMMAND

If the "script" command is executed without any options or arguments, the operating system simply spawns another shell for you. This script program functions as a normal shell, but because it is the child of the shell or script program from which the command was executed, it does not know what your home directory is. The "log" command terminates the child shell and returns control to the parent script.

The "script" command can also be executed with options only. This form of the command also spawns a script program that interacts with you. If used in the password file, the command should be executed with the 'l' option (see Options Available).

Finally, the "script" command can be executed with arguments or with both options and arguments. This form may be used, for example, to execute a "script" script for which you do not have execute permissions. Either of the following commands executes the file "scriptfile":

```
script scriptfile
script <scriptfile
```

"script" first checks to see that the file specified as an argument is actually a file that contains commands. If it is not, "script" executes it only if you specify the 'c' option (see Options Available).

Arguments

<argument_list>	A list of arguments to pass to the script command. Each element in the argument list consists of a command name followed by the appropriate arguments and options. The elements in the list must be separated by a valid command separator (";", "&", "&&", or "||"). If any separator characters are used, the entire argument list must be enclosed in s or double quotation marks.

Options Available

Options specified to the script program must appear immediately after the name "script" on the command line, so that they are not confused with options that pertain to the arguments passed to the script.

- a Start execution with the "sabort" attribute off.
- b Ignore control-C and control-\.
- c Process the argument list as a command.
- l Run as a login shell. A login shell tries to find the name of the user's home directory by looking in the file ".home?". It also automatically executes the file ".startup" in the working directory.
- v Start execution with the verbose attribute on.
- x On the next command, do not fork unless necessary. This option is used only when calling a script program from another program.

NOTE

It is impossible to specify a null string as an argument to a command because the script program removes null strings from the command line.

ERROR MESSAGES

Built-in commands may not use pipes.

Input to or output from the script built-in commands ("chd," "dperm," "jobs," "log," "login," and "wait") may not be routed through a pipe.

Cannot execute "<cmd_name>".

The operating system was unable to execute the specified command. Either the command does not exist or you do not have execute permission.

Cannot initialize tables.

This error, which should not occur, is usually indicative of a hardware failure. If it does occur, contact your Tektronix field office.

Cannot open I/O redirection file.

The operating system returned an error when the script program tried to open the file specified for I/O redirection. Most probably, the path specified cannot be followed (one of the directories does not exist) or you do not have the permissions necessary for opening the file. This message is preceded by an interpretation of the error produced by the operating system.

Cannot open pipe.

The operating system returned an error when the script program tried to open the specified pipe. This message is preceded by an interpretation of the error produced by the operating system.

Error opening a file.

The operating system returned an error when the script program tried to open the specified file. This message is preceded by an interpretation of the error produced by the operating system.

Error reading a file.

The operating system returned an error when the script program tried to read the specified file. This message is preceded by an interpretation of the error produced by the operating system.

Error writing a file.

The operating system returned an error when the script program tried to write to the specified file. This message is preceded by an interpretation of the error produced by the operating system.

I/O redirection conflict.

You tried to redirect standard input, standard output, or standard error to more than one place.

I/O redirection error.

The operating system returned an error when the script program tried to perform the specified I/O redirection. This message is preceded by an interpretation of the error produced by the operating system.

Memory overflow.

There is not enough memory available to perform the specified command. Most probably, the expansion of the matching characters used on the command line, for which many matches were possible, caused the error.

Missing "]" or invalid character range.

Either the right square bracket is missing from the specification of a range of matching characters, or the range specified is invalid.

No matching file names found.

Matching characters appear on the command line, but no file names match the specified pattern.

Parenthesis usage error.

The parentheses used on the command line are unbalanced.

Too many tasks.

The script program tried to fork, but too many tasks were running at the time. The limit to the number of tasks allowed either to the individual user or to the operating system as a whole was reached.

Unknown error.

This error should not occur. If it does, contact your Tektronix field office.

Unrecognized argument to built-in command.

The argument specified is not a valid argument to the built-in command in question.

Unterminated string.

The quotation marks used on the command line are unbalanced.

SEE ALSO

chd
dperm
jobs
log
login
time
wait

shell

DESCRIPTION

"shell" is an interactive command language that gives you many conveniences when working with the 4404 operating system. When using "shell" as the command language, you can do command line editing, as certain editing keys are defined as in the EMACS text editor.

Editing and History

"shell" remembers a limited number of commands. You can use the shell command "history" to retrieve a list of commands that "shell" accepted. You can then use control (or function) keys to recall and modify commands.

You enter commands one character at a time, editing the command line (either with backspace and re-typing or with the command editor) and press the return key to execute the command.

Table 2-1 shows the keys or key sequences associated with the "shell" editing functions and a brief description of those functions.

Table 2-1

"SHELL" EDITING KEYS AND FUNCTIONS

Key	Function	Description
^P	up	Recalls the previous command with the same prefix.
^F	right	Moves the cursor right one character.
^B	left	Moves the cursor left one character.
^D	erase character	Erases the character at the cursor.
^H or DEL	backspace	Erases the character preceding the cursor.

ESC-F	word right	Moves the cursor to the right to the start of the next word.
ESC-B	word left	Moves the cursor to the left to the start of the nearest word.
ESC-D	erase word	Erases to the end of the word at or following the cursor.
ESC-H or ^W	erase back word	Erases the word before the cursor.
^A	begin line	Moves the cursor to the beginning of the line.
^E	end line	Move the cursor to the end of the line.
^K	erase to end	Erase characters from the cursor to the end of the line.
^U	erase line	Erase (or restore) the entire line.
^T	transpose	Transpose the previous two characters.
^L	redisplay	Redisplay the current line.
^Q	quote	Enters the key value of the following key.
RET or LF	return	Executes the command.

When editing, the characters you insert will appear at the cursor position and the following characters will shift to the right.

The most commonly command used with "history" is "up." If you do not have the cursor positioned at the start of a line, successive calls to "up" recall only commands that begin with the same non-blank character string as that preceding the cursor. For example, if you have the cursor after the string "help" (where you had used the "help" command) pressing ^P will take you back to the previous command where you used "help."

ENVIRONMENT VARIABLES AND ALIASES

A list of name-value pairs called environment variables is kept by "shell." When "shell" encounters a string that it recognizes as an environment variable, it emits the value it has stored for that variable. You may define or modify an environment variable by writing a quoted string of the form: "name=value" to "shell." For example to define the variable COMMAND as /bin, type the string "COMMAND=/bin." Then, to change your working directory to /bin, type "chd $COMMAND."

You can delete environment variables with "unset," used as "unset COMMAND." The "set" command displays the currently listed environment variables.

Search Path

The environment variable "PATH" defines the search path for the directory containing the command. Each alternative directory name is separated by a colon. If the command name contains a "/", the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission, but is not a binary file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned and the script shell "script" is used to read and execute it. A command contained within parentheses is also executed in a subshell.

Variable Arguments

Variables may contain argument designators to extract arguments from commands (such as used when defining aliases). The argument designators are:

$0	The first word of the command (the command itself)
$n	The nth argument of the command
$^	The first argument of the command (equivalent to $1)
$$	The last argument of the command
$x-y	The range of arguments from x to y (such as $3-5)
$-y	Abbreviation of $0-y
$*	Abbreviation of $^-$ ($1 $2 ...$$)
$n*	Abbreviation of $n-$
$n-	Abbreviation of $n-($-1) (omits last argument)
$-	Abbreviation of $0-($-1) (omits last argument)

When evaluating aliases, these argument designators extract the arguments from the command line to pass to the aliased commands.

Aliases

"shell" maintains a list of aliases, or command redefinitions. When you enter a command line, "shell" checks the first word of the command to see if it is an alias. If so, "shell" executes the text of the alias and can use argument designators to extract the arguments to the aliased command.

You can create or modify an alias with the "alias" command. You can delete an alias with the "unalias" command. You can see the currently defined aliases by entering the "alias" command without any arguments.

For example, if a Unix<tm> programmer were to want the command "ll" to perform the action of the operating system command "dir +1," that person could create that alias by typing (without the double quotes) "alias ll 'dir +1 $*'". Then typing "ll /bin" would have the same effect as typing "dir +1 /bin."

Function Keys

The function keys and joydisk are represented by special environment variables. By defining these variables, you can cause the joydisk and function keys to perform actions. When you press a function key or the joydisk, "shell" echoes the string defined for that variable.

You can insert special characters into function key and joydisk variable definitions by using the quote character, ^Q. The following

The twelve function key variables are $F1 - $F12 and the joydisk variables are $JOYUP, $JOYDOWN, $JOYLEFT, and $JOYRIGHT. The "Break" key is bound to the variable $BREAK, and the arrow key (upper right of keyboard) is bound to $ARROW and $SARROW for the shifted arrow key.

COMMAND SYNTAX

A command is either a simple-command or a list.

A simple-command is a sequence of non blank words separated by blanks (a blank is a tab or a space). The first word specifies the name of the command to be executed. Except as later specified, the remaining words are passed as arguments to the invoked command. (The command name is passed as argument 0.)

A list is a sequence of one or more pipelines separated by ";" or "&", and optionally terminated by ";" or "&". ";" and "&" have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding pipeline to be executed without waiting for it to finish. Newlines may appear in a list, instead of semicolons, to delimit commands.

A pipeline is a sequence of one or more commands separated by "|". The standard output of each command but the last is connected by a pipe to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

Command Substitution

The standard output from a command enclosed in a pair of back quotes (``) may be used as part or all of a word; trailing newlines are removed.

"Wild Card" Characters

Following substitution, each command word is scanned for the characters "*", "?" and "[". If one of these characters appears, the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character "." at the start of a file name or immediately following a "/", and the character "/", must be matched explicitly.

The special characters match in this manner:

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed. A pair of characters separated by "-" matches any character lexically between the pair.

An additional special character is the tilde. When a tilde is the first character in a filename, "shell" expands it by replacing it by the home directory of the named user. For example, if user sandra has a home directory (defined in the password file) of /public /sandra, the filename "~sandra/file" expands to "/public/sandra/file."

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted.

";" "&" "(" ")" "newline" "space" "tab"

A character may be quoted by preceding it with a "\". "\newline" is ignored. All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted. Inside double quotes (" ") parameter and command substitution occurs and "\" quotes the characters "\", "'",'"', and "$".

Execution

Each time a command is executed, the above substitutions are carried out.

You can run commands in the background by inserting a "&" as either the first or last nonblank character on a command line. "shell" prints the name and process ID for each background task when it begins, and again when it terminates.

You can group commands for a subshell with parentheses, put the subshell in the background by following the closing parentheses with "&," and redirect I/O for the subshell.

You can time execution of a command by using "%" as the first or last nonblank character on a command line. "shell" prints the real, user, and system times for the command's execution.

To quickly access the script shell, "script," use "!" as the first non-blank character on a line. To pass the remaining characters to "script" uninterpreted, use the +c option.

Redirecting Input and Output and Error

To redirect standard output, use ">" and ">>." ">" directs standard output of a preceding command into the filename following it, writing over an old file. ">>" appends the standard output of a preceding command into the filename following it.

To redirect standard input into a command, follow the command with "<" in front of the command that will generate the input for the first command.

To redirect standard error, use "^," and "^^" as you would standard output redirection. You can combine redirection of standard input, output, and error to a file by using a combination of symbols. For example you can redirect both standard error and output to the file "temp" with "^>temp." You can also connect both standard output and error to a pipe with "^|."

SUMMARY OF "SHELL" COMMANDS

Table 2-2 lists the commands (followed by a brief description) that are part of "shell." You cannot redirect I/O for these commands.

Table 2-2

"shell" COMMANDS

Command [arguments]	**Description**
alias [name][string]	With no arguments, prints the names of all defined aliases. With one argument, prints the associated alias. With two arguments, the second argument is defined to be an alias for the first.
chd [arg]	Change current directory (default to user's home directory)
dirs	Lists the current working directory and the directory stack. Lists the directory stack.
dperm [u-rwx][o-rwx]	Sets default permissions for file creation.
history	Displays saved command history
jobs	Lists currently executing background jobs for present user.
login [arg]	Terminate this interactive session and start the login process.
logout	Terminate this interactive session.
exit	Terminate a subshell.

popd	Changes the working directory to the one whose name is on the top of the directory stack.
pushd [dir]	Pushes the name of the working directory on the directory stack and changes to the specified directory. With no argument, this command exchanges the top of the directory stack and the current working directory.
set [file]	Without an argument, "set" displays the current state of the shell and the values of the defined environment variables. If you specify a file, it executes the commands in it as if you had typed them. Use this option to set environment variables and the user file creation mask. "set" terminates an input line and cannot be used as an alias.
unalias [name]	Deletes the named alias from the set of aliases.
wait	Waits for all background processes to terminate and reports their termination status. If the "wait" command is interrupted, then a list of currently active processes is displayed.

SYNTAX

shell [+l][+h=<filename>][+c <string>][+i][<filename>]

DESCRIPTION OF THE "SHELL" COMMAND

If you call "shell" with no arguments, it spawns a subshell with which you then interact until you issue either the "exit" or "logout" commands. This shell executes commands in the file ".shellbegin" in your home directory, but does not store the name of your home directory. When you exit the subshell, control returns to the parent shell.

Options and Arguments

l — The "l" option tells "shell" to run as a login shell. This option causes shell to execute commands from the files ".login" and ".shellbegin" (in your home directory) when it begins execution, and from the file ".logout" when it terminates. The "exit" command terminates a subshell, use "logout" to end a session with the login shell.

h=<filename> — This option causes "shell" to initialize its state from that saved in <filename>. When "shell" terminates it saves its history, environment variables, and aliases into this file. Without this option, "shell" reads and writes its state into the file ".shellhistory" in your home directory. To prevent state recovery and saving, use "none" as the <filename> (+h=none).

<filename> — If "shell" is followed by a filename without the "c" or "i" options, it assumes that the file is a command script. "shell" passes control and the argument to the script shell, "script."

c <string> — The "c" option causes "shell" to assume the next string of characters is a shell command, to execute that command and then terminate.

i <filename> — The "i" option causes "shell" to process the commands contained in <filename> and then terminate, rather than passing the commands to "script."

DIAGNOSTICS

"shell" gives error messages similar to other messages detailed in this manual whenever directories and files cannot be opened, whenever it detects a syntax error, and when it reaches its memory limits.

LIMITS

"shell" has the following limits:

- 256 environment variables
- 30 saved commands (history)
- 16 entries on the directory stack
- 128 characters per command line
- Command expansion cannot exceed 512 arguments and 5120 characters

SEE ALSO

script

status

Display the status of running tasks.

SYNTAX

```
status [+alswx]
```

DESCRIPTION

The "status" command reports, to standard output, the status of tasks running on the system. By default, this report does not include shell or login programs and is restricted to tasks belonging to the user who executes the command. The command is not always completely accurate due to the dynamic nature of the operating system. By default, the "status" command reports on the following parameters:

Task-id — The number assigned to the task by the operating system.

Mode — Indicates whether the task is in memory ('c') or has been swapped to the disk ('s').

tty — The number of the terminal from which the task originated. An "xx" in the field indicates that no terminal is associated with the task.

Prio — If the entry in this field is a number, it indicates the priority of the task. A higher number indicates a higher priority. Other priorities are described in Table 2.3.

Table 2-3

POSSIBLE TASK PRIORITIES

Priority	Meaning
buf	Waiting for a system buffer.
disk	Waiting for some disk activity.
file	Waiting for some file activity.
halt	Halted by another task.

in	Waiting for input from the terminal.
out	Waiting for output to the terminal to end.
pipe	Waiting for pipe data (usually input)
upd	Updating an fdn.
slp	Sleeping (not executing).
swap	Being swapped to or from the disk.
sys	Highest possible priority.
wait	Waiting for another task to end.

Time — If the command is "System", this parameter is the amount of unused CPU time since the system was booted. Otherwise, it is the total CPU time that a particular task has used.

Command — The command which originated the task. By default, the "status" command shows the first thirty-five characters of the command line; the rest are truncated. The command "System" is the operating system. The command "/etc/init" executes the login program. If the "status" command cannot determine what was on the command line, this field contains the entry "???".

Options Available

a — List all tasks on the system, not just those belonging to the user.

l — Produce a more detailed description of the status of each task.

s — Produce a statistical summary of the use of the operating system.

w[=<num>] — Wait <num> seconds after reporting the status; then produce another report. The command repeats 100 times. The default is thirty seconds.

x List all tasks (a normal listing does not include shell programs, the "System" command, or the command "/etc/init").

If the user specifies the 'l' option, the following additional items are included in the report:

Status The status of the task. Possible values include run (task is running), sleep (task is waiting for something to happen), and term (the task has terminated).

User The user name of the person who owns the task. If two or more user names share the same user ID, "status" uses the name that appears first in the password file.

Parent The task ID of the parent task. If the parent task in no longer active, the ID shown in this field is 1.

Size The amount of memory that the task is using.

Res A rough measure of the amount of time a task has been in memory or swapped out to the disk. Each unit represents four seconds. The largest number that is ever displayed is 255. This number is set to 0 whenever a task is swapped into or out of memory.

If the user specifies the 's' option, the following statistics are included in the report. They represent activity on the system since the time the system was booted.

Total block I/O transfer attempts.

The number of times the system has tried to access a disk block in the cache.

Total disk I/O operations.

The number of times the system has had to access the disk. This statistic does not include swap operations.

Total blocks freed.

The number of blocks that have been released from a file to the free list. If the same block has been released more than once, each release is counted.

total system calls
total PAGE IN operations
total PAGE OUT operations
total pages stolen

EXAMPLES

1. status +s
2. status +alxw=15

The first example displays the default information about the status of all tasks except shell programs that belong to the user. A summary of the use of the operating system is included in the output.

The second example displays detailed information about the status of all tasks on the system. It waits fifteen seconds, then issues another report. The command repeats 100 times unless the user interrupts it by typing a control-C.

stop

Stop the system and prepare to shut off the power or reset.

SYNTAX

stop

DESCRIPTION

The command "stop" terminates any background processes, closes open files, flushes buffers to the disk, and does the general housekeeping necessary to perform an orderly system shut-down.

You should always run "stop" before turning off the power to the 4404 or pressing the Reset Button.

EXAMPLES

stop

This is the only form of this command.

MESSAGES

When "stop" is finished, it prints the message:

... System shutdown complete ...

At this point, the system has been completely shut down and it is safe to turn off the power or to reset the system.

strip

Remove the symbol table from an executable binary file.

SYNTAX

strip <file_name_list>

DESCRIPTION

The "strip" command removes the symbol table from an executable binary file.

Arguments

A list of files to process.

EXAMPLES

strip testprog

This example removes the symbol table from the executable binary file "testprog".

ERROR MESSAGES

Error creating "<file_name>": <reason>

The operating system returned an error when "strip" tried to create the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error opening "<file_name>": <reason>

The operating system returned an error when "strip" tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Error unlinking "<file_name>": <reason>

The operating system returned an error when "strip" tried to unlink the specified file. This message is followed by an interpretation of the error returned by the operating system.

File "<file_name>" cannot be located.

The specified file does not exist.

File "<file_name>" is a device or a directory.

The specified file is not a regular file.

tail

Print a specifiable number (up to 250) of characters from the end of a text file.

SYNTAX

```
tail file [n]
```

DESCRIPTION

This utility prints the last "n" characters in a text file. If "n" characters from the end of the file happens to fall in the middle of a line, the line will be preceded by "..." to show that only a part of the line has been printed. Whole lines are printed as they appear in the file.

Special characters such as carriage returns and tabs are counted as part of the "n" characters.

Arguments

file	The file from which characters are to be printed.
n	The number of characters from the end to start printing. The default is 250 characters. If "n" exceeds the number of characters in the file, the whole file is printed.

EXAMPLES

1. tail .shellbegin
2. tail testfile 30

The first example will print the last 250 characters of ".shellbegin", or the entire file if it contains less than 250 characters.

The second example prints the last 30 characters from the file "testfile."

SEE ALSO

head

touch

Set the time of the last modification of a file to the current date and time.

SYNTAX

touch <file_name_list>

DESCRIPTION

The "touch" command sets the time of last modification for the specified file to the current date and time. The user must have read and write permission in a file in order to "touch" it. This command is often used in conjunction with the "update" command. It is also useful for correcting the last modification time of a file which was created or modified when the system time was incorrect.

Arguments

<file_name_list>	A list of the names of the files to modify.

EXAMPLES

touch letter memo

This example changes the modification time of the "letter" and "memo" files to the current date and time.

ERROR MESSAGES

Error seeking to beginning of file "<file_name>": <reason>

The operating system returned an error when "touch" tried to seek to the beginning of <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Error touching "<file_name>": <reason>

The operating system returned an error when "touch" tried to change the last modification time of <file_name>. This message is followed by an interpretation of the error returned by the operating system.

File "<file_name>" does not exist!

The "touch" command could not find <file_name> in the file system.

SEE ALSO

date
update

update

Process a set of files, performing the specified operation on each file if it is newer than the file it is compared to.

SYNTAX

update [<make_file_name>] [+q]

DESCRIPTION

The "update" command reads the specified "makefile", which must conform to a special format, and conditionally performs the command or commands in that file. By default, the "update" sends informative messages to standard output telling the user what it is doing. The command is most often used to recompile programs whose sources have been updated.

Arguments

<make_file_name>	The name of the file to read for instructions. This file must be in a special format (see Format of the "makefile"). The default is the file "makefile" in the working directory.

Format of the "makefile"

The "makefile" is composed of modules, each of which is terminated with a percent sign, '%', in column 1. A module itself is composed of up to two parts. The first part specifies the process that "update" is to perform. The format for this first part is as follows:

[<item-one>::<item_two>;]<command_sequence>

where <item_one> and <item_two> are the names of files; "::" is the "is newer than" operator; and the semicolon, ';', separates the file names from the command sequence.

The command sequence is composed of one or more operating system commands. The "update" command replaces any sequence of more than one space character with a single space. Multiple commands are separated by additional semicolons. If the commands do not fit on one line, the user must begin and end the sequence with an exclamation point, '!', which serves to delimit the entire command sequence. If the first portion of the module uses more than one line, the second exclamation point marks the boundary between the first and second portions of the module. The command sequence is executed if <item_1> is newer than <item_2>.

The user may substitute an ampersand, '&', for any character or sequence of characters in <item_one>, <item_two>, or the command sequence. In such a case the "update" command substitutes for all ampersands the strings specified in the second portion of the module. If the second portion of the file is absent, no command sequence is performed. This portion consists of one or more lines, each of which contains a single string to substitute for the ampersands. The "update" command replaces each occurrence of an ampersand with the string on the first line of the second portion of the module and performs the command sequence if <item_one> is newer than <item_2>. It then replaces all ampersands with the string from the second line, continuing in this fashion until it reaches the end of the second portion of the module (marked by a percent sign in column 1).

If the file represented by <item_two> does not exist, "update" considers that <item_one> is newer than <item_two>. If the file represented by <item_one> does not exist, or if neither the file represented by <item_one> nor <item_two> exists, <item_one> is not considered to be newer than <item_two>.

For instance, consider the following "makefile":

```
&::&.b;asm & +sly
file_1
file_2
   .
   .
   .
file_n
%
```

An "update" command which references this file makes the following translation:

If "file_1" is newer than "file_1.b", execute the command "asm file_1 +sly".

If "file_2" is newer than "file_2.b", execute the command "asm file_2 +sly".

It continues in this fashion until "file_n" is processed. The percent sign in column 1 marks the end of the module, and because it is the only module in the file, the "update" command terminates.

More than one set of commands can be processed with a single "makefile" if the user includes more than one module in the file.

Options Available

q Do not send informative messages to standard output.

NOTES

- The "chd" command has no effect in a "makefile".
- The "update" command always tries to substitute the strings strings specified in the second portion of a module for all ampersands which appear in the first portion. Thus, the command sequence itself cannot contain an ampersand. Consequently, tasks specified in a "makefile" cannot be executed in the background (although the "update" command itself may be sent to the background).

ERROR MESSAGES

*** Can't access Makefile "<file_name>" aborted!

The operating system returned an error when "update" tried to open <file_name> for reading. Most probably, the file specification is incorrect, the file does not exist, or the user does not have read permission for the file.

a*** Error: Command too complicated.
<command_sequence>

After substitution for the ampersands has taken place, the command sequence is too long (the limit is 512 characters).

*** Error: Pattern too complicated.
<command_sequence>

The pattern for the command sequence (before substitution for ampersands takes place) is too long (the limit is 512 characters).

Makefile syntax error aborted

The "update" command was unable to interpret the "makefile".

Syntax: update [<make_file_name>] [+q]

The "update" command requires exactly one argument. This message indicates that the argument count is wrong.

Unknown option: <char>

The option specified by <char> is not a valid option to the "update" command.

SEE ALSO

touch

wait

Wait for a background task to complete before accepting any more input.

SYNTAX

```
wait [<task_ID>]
wait any
```

DESCRIPTION

The "wait" command, which is part of the shell program, tells the shell program not to accept any more commands until the specified background task is complete. The termination status of the task is reported when the task is complete. If the user does not specify a task ID, the shell program waits for all active background tasks that are children of the shell program that issued the "wait" command to finish before accepting any new commands. The user may interrupt the "wait" command with a control-C.

Arguments

<task_ID>	The ID of the task to wait for. The shell program reports the ID when it sends a task to the background. The ID may also be obtained by executing either the "jobs" or the "status" command.
any	If the user specifies the argument "any", the shell program waits for any one background task that is one of its children to finish before accepting a new command.

EXAMPLES

1. wait 495
2. wait
3. wait any

The first example tells the shell program to accept no further commands until task 495 is complete.

The second example tells the shell program to accept no further commands from the user until all background tasks belonging to that shell program are complete.

The third example tells the shell program to accept no further commands from the user until one background task belonging to that shell is complete.

ERROR MESSAGES

No tasks running in the background.

The shell program has no tasks running in the background.

Specified task not running in the background.

The task specified either is not a child of the current shell program or does not exist.

SEE ALSO

jobs
shell
status

Section 3

"SYSTEM" UTILITIES

These utilities are generally reserved for the user logged in as "system." They tend to be either powerful utilities, with great potential for misuse, or utilities that should be reserved to a limited number of users where many accounts are set up.

User "system" generally has the directory "/etc" defined in the search path, and needs only enter the name of the utility to invoke it. The full path name is given here, however, to emphasize the special purpose of these utilities.

adduser

Add a new user to the system.

SYNTAX

/etc/adduser <user_name>

DESCRIPTION

The "adduser" command is used to add a new user to the system. The specified user name must be unique to the system. It must be between one and eight letters long. All letters must be lower-case. Only the "system" user may invoke this command.

The "adduser" command performs the following tasks:

1. Adds the new name to the end of the password file, "/etc/log/password".

2. Assigns a user ID to the user.

3. Creates a home directory owned by the new user with "rwxr-x" permissions. The name of this directory is "/<user_name>".

The "system" user or the new user should use the "password" command to ensure protection of the new user's files.

Arguments

<user_name>	A unique name assigned to the new user for use in response to the login prompt.

EXAMPLE

```
/etc/adduser chris
```

This example adds the user name "chris" to the bottom of the file "/etc/log/password", assigns a user ID, and creates the directory "/chris"--which is owned by "chris" and has permissions "rwxr-x".

ERROR MESSAGES

Error adding "<user_name>" to password file: <reason>

The operating system returned an error when "adduser" tried to add "<user_name>" to the password file. This message is followed by an interpretation of the error returned by the operating system.

Error assigning owner to "/<user_name>": <reason>

The operating system returned an error when "adduser" tried to make the specified user the owner of the file "/<user_name>". This message is followed by an interpretation of the error returned by the operating system.

Error creating "/<user_name>": <reason>

The operating system returned an error when "adduser" tried to create the file "/<user_name>". This message is followed by an interpretation of the error returned by the operating system.

Error creating "." file: <reason>

The operating system returned an error when "adduser" tried to create the file ".". This message is followed by an interpretation of the error returned by the operating system.

Error creating ".." file: <reason>

The operating system returned an error when "adduser" tried to create the file "..". This message is followed by an interpretation of the error returned by the operating system.

Name must be 1 to 8 lowercase letters.

The specified user name must be between one and eight letters long. All letters must be lowercase.

Syntax: /etc/adduser <user_name>

The "adduser" command expects exactly one argument. This message indicates that the argument count is wrong.

The name "<user_name>" is already in use.

The specified user name must be unique to the system.

You must be system manager to run "adduser".

Only the "system" user may execute the "adduser" command.

SEE ALSO

deluser
password
perms

blockcheck

Check the integrity of the allocation of all blocks used in files and of the free list on the specified device.

SYNTAX

/etc/blockcheck <dev_name>

DESCRIPTION

"blockcheck" checks the integrity of the block allocation used in the files and free list on the specified device. It locates problems such as duplicate blocks, missing blocks, and invalid block addresses.

This command is primarily intended for use by the "diskrepair" utility, which calls it. It may also be used on its own as a diagnostic utility; however, "blockcheck" can only check the disk; it cannot repair it. If "blocheck's" output suggests that the disk is damaged, use "diskrepair" on the disk.

You should only use "blockcheck" if no other tasks are active on the system; otherwise, the results are unpredictable.

Arguments

<dev_name>	The name of the device to check. It must be a block device.

EXAMPLES

/etc/blockcheck /dev/floppy

This example checks the integrity of the the allocation of blocks on the floppy disk.

SEE ALSO

devcheck
diskrepair
fdncheck

deluser

Remove a user from the system.

SYNTAX

/etc/deluser <user_name> [x]

DESCRIPTION

The "deluser" command removes the specified user from the system. It removes the corresponding entry from the file "/etc/log/password" and destroys files and subdirectories in the user's home directory that are owned by that user. It also deletes the home directory itself if it is empty after all the deletions are complete. Only the "system" user may execute this command.

Arguments

<user_name> The name of the user to delete from the system.

Options

x Delete the user, but do not delete the user's files from the system.

EXAMPLES

/etc/deluser chris

This example deletes the line containing the entry for the user name "chris" from the file "/etc/log/password". It also deletes all files and subdirectories in the directory "/chris," as well as that directory itself.

CAUTION

This command should be used with great care as it may recursively descend the user's directory tree, deleting all files within it that are owned by the specified user.

ERROR MESSAGES

Cannot delete a user with an ID of 0 or 1.

The "deluser" command cannot delete user ID 0 (system) or 1 (public).

Cannot execute "remove".
<user_name>" not removed from system.

The "remove" command, which is called by "deluser" is not in "/bin" or "/bin". The command aborts without editing the password file.

Name must be 1 to 8 lowercase letters.

The specified user name must be between one and eight letters long. All letters must be lowercase.

Syntax: /etc/deluser <user_name>

The "deluser" command expects exactly one argument. This message indicates that the argument count is wrong.

<user_name>" is not in the password file.

The file "/etc/log/password" does not contain an entry for the specified user name.

You must be system manager to run "deluser".

Only the "system" user may execute the "deluser" command.

SEE ALSO

adduser
remove

devcheck

Check a device for I/O errors.

SYNTAX

/etc/devcheck <dev_name> [+fsv]

DESCRIPTION

The "devcheck" command checks the specified device for I/O errors. As it checks the device, it prints informative messages, which tell the user what part of the device is being checked. It always checks the boot sector and the system information record (SIR). By default, it also checks the fdn space, the swap space, and the volume space.

Every time it finds a bad block, it prints a message giving the address of the block in hexadecimal. When it is finished, "devcheck" prints a message reporting the total number of bad blocks on the disk.

If a floppy disk contains one or more bad blocks, it should probably be discarded. If a hard disk contains one or more bad blocks, it should be reformatted with the addresses of all bad blocks placed in the file ".badblocks". It is wise to run this command immediately after formatting a disk.

Arguments

The name of the device to check. It must be a block device.

Options

- Check only the fdn space.
- Check only the swap space.
- Check only the volume space.

EXAMPLES

1. /etc/devcheck /dev/floppy
2. /etc/devcheck /dev/floppy +v

The first example checks the entire disk in the floppy drive for I/O errors.

The second example checks the boot sector, the SIR, and the volume space of the disk in the floppy drive for I/O errors.

MESSAGES

Badblocks file too large - continuing without list.

"Devcheck" cannot read a ".badblocks" file that has more than 138 bad blocks in it. Currently, this theoretical limitation on the number of bad blocks is unlikely to present a practical limitation. The number of bad blocks on a disk should not even approach 138.

Can't open character device '<dev_name>'.

The "devcheck" command cannot open the character device which corresponds to the block device specified on the command line. Most probably, either the device does not exist or the user does not have the permissions necessary to open it. In such a case the command continues, but it may report the blocks in the file ".badblocks" as bad.

Can't read '.badblocks' file - continuing without list.

The "devcheck" command encountered an I/O error when it tried to read the file ".badblocks".

File '.badblocks' not found - continuing with check.

The device specified does not contain a file named ".badblocks", or due to damage in the logical structure of the disk, "devcheck" cannot locate the file.

ERROR MESSAGES

Can't open '<dev_name>'.

The "devcheck" command cannot open the device specified on the command line. Most probably, either the device does not exist or the user does not have the permissions necessary to open it.

File '<file_name>' is not a block device.

The "devcheck" command can only check a block device.

Unknown option '<char>' - option ignored.

The option specified by <char> is not a valid option to the "devcheck" command.

SEE ALSO

blockcheck diskrepair fdncheck

diskrepair

Check and, optionally, repair inconsistencies in the logical structure of a disk.

SYNTAX

/etc/diskrepair <dev_name_list> [+bfmnpqruv]

DESCRIPTION

The "diskrepair" utility checks the structure of the disk or disks specified in <dev_name_list>. The structure of the disk refers to the layout of and the connections among files, directories, free space, swap space, and other information that makes up the file system. Any inconsistencies in the structure are reported and, optionally, repaired. "Diskrepair" does not check or repair media errors (I/O errors).

Related Utilities

While it is operating, "diskrepair" calls two other utilities--"blockcheck" and "fdncheck", which are both located in the directory "/etc".

- "Blockcheck" is concerned with the allocation of blocks on the disk. It locates problems such as duplicate blocks, missing blocks, and invalid block addresses.
- "Fdncheck" is concerned with the directories on the disk. It locates problems such as unreferenced files, file names with invalid associated files, and so forth.

Major Modes of Operation

There are two major modes of operation, simple and verbose.

- The simple mode is selected by default; the verbose mode is selected by the 'v' option.
- In the verbose mode "diskrepair" reports all detected errors. In the simple mode it reports only those errors which require the deletion of files or of directory entries.
- If executed in simple mode, "diskrepair" issues a message upon completion which informs the user whether or not the disk is in need of repair. By default, all detected errors are automatically repaired (if possible).

Options

Two options ('n' and 'p') exist to alter the handling of errors.

- The 'n' option instructs "diskrepair" not to repair any errors. The 'p' option instructs "diskrepair" to prompt the user for permission to repair the errors it reports.

- In verbose mode the 'p' option causes "diskrepair" to prompt the user regarding all errors. In the simple mode, the user is prompted only for those errors which require the deletion of files or of directory entries; all other errors are automatically repaired without prompting.

NOTE

Most repairs result in a loss of data. The user can generally infer which data have been lost from the messages displayed.

- When using the command in simple mode (without the 'v' option), the user need not understand what types of checks are made by "diskrepair". The only decisions required are whether or not to delete the reported files. In verbose mode, much more information is given to the user.

While this document is not intended to give full details of this information, the following list shows most of the inconsistencies in disk structure for which "diskrepair" checks. First, however, a few definitions are in order.

Definitions

- A "file descriptor node" (or fdn) is an area on the disk which contains all the information the system needs about a file. There should always be one fdn per file on the disk.

- A 4404 directory entry is simply a file name and a pointer to the proper fdn. There may be multiple directory entries pointing to the same fdn (multiple names for the same file).

- Each pointer to an fdn is called a "link" to that file. If there is a file with no links, it is considered to be "unreferenced". "Out-of-range" refers to a pointer to a disk block or to an fdn which is beyond the valid number of blocks or fdns for the disk being tested.

Inconsistencies

Here now, is a partial list of inconsistencies that "diskrepair" checks for:

- Blocks duplicated in files or free list
- Out-of-range blocks or fdns
- Missing blocks
- Bad free list
- Unreferenced files
- Inactive fdns
- Unknown fdn type
- Incorrect link counts
- Incorrect free block or free fdn count
- Invalid sizes in System Information Record

Unreferenced Files

These are handled in one of two ways:

1. An attempt is made to give the file a name by putting it into the directory "lost+found" in the root directory of the disk being tested. The name given to the file is of the form "file<fdn>", where <fdn> represents the fdn number of the file.

   In order for this procedure to work, the directory "lost+found" must already exist on the disk being checked, and it must have room for the entry. The program "crdisk" creates this directory, but if for any reason it has been deleted, the user should recreate it before running "diskrepair". The user must must also create empty slots for entries by creating a number of files and then deleting them.

2. If it is not possible to put the unreferenced file into the "lost+found" directory (because there is either no directory "lost+found" or no room in it), "diskrepair" deletes the file (or prompts for permission to delete it if 'p' was specified).

Fdn Error Data

If an error is associated with an fdn, a display of pertinent data from that fdn is printed. The display includes the fdn number of the file, its size in bytes, its owner, the time of its last modification, and one of the following types:

b = block device
c = character device
d = directory
f = file
i = inactive
u = unknown

The "diskrepair" utility should generally be run only on an otherwise inactive system. It should never be run on an active disk. If the "n' option is not specified (the disk may be written to), "diskrepair" attempts to unmount the disk being tested. If the disk being tested is the system disk, and if a repair is made which requires writing to the System Information Record (block number 1), "diskrepair" stops the system upon completion and issues an appropriate message instructing the user to reboot the operating system. This procedure is necessary to prevent conflicts between the written data and similar data kept in memory.

Options

'b' Option	The 'b' option instructs 'diskrepair' to run only the 'blockcheck' portion of the utility. This procedure is often considerably faster, but still provides a fairly complete assessment of the validity of the disk structure.
'f' Option	The 'f' option instructs "diskrepair" to run only the "fdncheck" portion of the utility. This option is useful if a problem is suspected in the directory structure, but the result is by no means a thorough check of the structure of the disk.
'm' Option	The operating system maintains a list of blocks available for use called the free list. A missing block is any block in the volume space which is not a part of any file and is not in the free list. The existence of such blocks is a harmless error in the structure of the disk.

"Diskrepair" generally places missing blocks in the free list. The 'm' option, however, instructs "diskrepair" not to rebuild the free list solely on account of missing blocks. This option reduces the time required for "diskrepair" to run if missing blocks are the only problem in the free list.

'n' Option — The 'n' option tells "diskrepair" to report all errors but to make no attempt to fix them. Therefore, "diskrepair" opens the device for reading only. This option is useful for checking the structure of a disk without risking the loss of data during repairs.

'p' Option — If the user specifies the 'p' option, "diskrepair" reports each error, followed by a prompt requesting permission for the proposed repair. All prompts require an answer of either 'y' ("yes") or 'n' ("no").

NOTE

Many repairs result in the loss of data. (You can generally infer what has been lost from the messages "diskrepair" displays.) Judicious use of the 'n' and 'p' options not only allows you to assess the damage to the disk and decide which information you are willing to sacrifice during the repair process; it also gives you the opportunity to try to salvage the data before repairing the disk.

'q' Option — This option inhibits certain warnings and messages from "diskrepair". Several conditions exist which, while not technically errors in disk structure, may cause problems. These conditions usually result in a warning message; the 'q' option inhibits them.

'r' Option — By default, if "diskrepair" finds that the free list is in error, it rebuilds it. The 'r' option instructs "diskrepair" to rebuild the free list whether or not it contains errors. This option is useful if the free list is known to be bad or if the user wants to reduce fragmentation within the list.

'u' Option — The 'u' option generates a report on the block usage of the specified device. This report is printed at the end of the "diskrepair" operation, and contains statistics on the following: (1) the number of each type of file in the file system and the total number of files in the system; (2) the number of unused blocks and the number of used blocks, including a breakdown of how the used blocks are allocated; (3) the number of free fdns and the number of fdns in use.

'v' Option — "Diskrepair" operates in one of two modes: simple or verbose. Simple mode is selected by default; verbose mode is selected by the 'v' option. In simple mode, "diskrepair" reports only those errors which require the deletion of either files or directory entries. In verbose mode, all errors are reported. In addition, informative messages are printed describing what phase "diskrepair" is performing.

In verbose mode the 'p' option causes "diskrepair" to prompt for permission regarding all errors. In simple mode the user is prompted only for those errors which require the deletion of either files or directory entries; all other errors are automatically repaired without prompting.

EXAMPLES

1. /etc/diskrepair /dev/disk
2. /etc/diskrepair /dev/disk +n
3. /etc/diskrepair /dev/floppy +pv
4. /etc/diskrepair /dev/floppy +ru
5. /etc/diskrepair /dev/disk +mq

The first example checks the logical structure of the system disk. By default, "diskrepair" tries to fix every error it encounters. These repairs may result in the loss of data from the disk.

The second example checks the logical structure of the system disk, reports those errors which require the deletion of either files or directory entries, but performs no repairs.

The third example checks the logical structure of the disk in the floppy drive. "Diskrepair" reports all errors it finds and prompts for permission before making any repairs.

The fourth example checks the logical structure of the disk in the floppy drive. "Diskrepair" rebuilds the free list no matter what and prints a summary of block usage when it is finished.

The fifth example also checks the logical structure of the disk in the floppy drive. It does not rebuild the free list solely on account of missing blocks; neither does it print the warnings and messages which result from problems not technically errors in the structure of the disk, but which may cause problems.

NOTES

"Diskrepair" cannot solve all the problems your disk may have. For example, it cannot fix physical media problems. As for problems with the logical structure of the disk, "diskrepair" can only repair an error if the damaged information is redundant -- that is, if there is some way of determining what the information should be.

"Diskrepair cannot, for instance, fix a badly damaged SIR; nor can it repair a disk if the root directory is severely damaged. It is therefore imperative that up-to-date backups of all important files be maintained.

ERROR MESSAGES

Blockcheck terminated abnormally.

"Blockcheck" received a program interrupt from the operating system. The user cannot determine the source of such an error; however, it is not indicative of a problem with either "diskrepair" or the device. "Diskrepair" should be rerun, for the problem may not recur.

Can't call /etc/blockcheck.

"Diskrepair" cannot read or execute the file "/etc/blockcheck".

Can't call /etc/fdncheck.

"Diskrepair" cannot read or execute the file "/etc/fdncheck".

Can't read System Information Record.

The SIR is so badly damaged physically that "diskrepair" cannot read it. The user may be able to salvage some information from the disk, but must eventually reformat it.

Can't stat root.

"Diskrepair" cannot read the fdn which describes the root directory. The user may be able to salvage some information from the disk, but must eventually reformat it.

Can't stat std. output.

"Diskrepair" cannot read the fdn of whatever file is opened as standard output. The user should rerun "diskrepair" with "/dev/console" as standard output.

Conflicting options.

The options specified on the command line conflict with each other.

Device is busy.

Any alterations that "diskrepair" makes must be made when the disk is not in use. Therefore, "diskrepair" determines whether or not the specified disk is mounted, and, unless the user specifies the 'n' option, it tries to unmount a mounted disk before proceeding. This error message means that either some user's working directory is on the specified disk or some task is accessing a file on that disk.

Disk needs repair!

The structure of the disk is not logically sound. The user should rerun "diskrepair" to correct the problems.

Error reading block <block_num>.

Error reading fdn <fdn_number> in block <block_num>.

Error writing block <block_num>.

Error writing fdn <fdn_num> in block <block_num>.

"Diskrepair" encountered a physical error on the disk. If either the 'p' or 'n' option is in effect, "diskrepair" prompts for permission to continue. If the user chooses to continue when the

'n' option is not in effect, the results are entirely unpredictable. They depend on precisely which block is damaged. Continuing with "diskrepair" may cause further damage to the disk, but in some cases, it may be the desired course of action.

NOTE

The first time "diskrepair" reports an I/O error, answer "no" to the offer to continue and immediately rerun "diskrepair". It is possible, though unlikely, that the I/O error is a soft one and will not recur.

Error updating SIR. Disk is bad!

"Diskrepair" encountered an I/O error when it tried to make the necessary changes in the SIR. The user should try again to execute "diskrepair". If the error persists, the user cannot salvage any of the data on the disk.

/etc/blockcheck is invalid.

The version of the "blockcheck" command is not the correct one.

/etc/fdncheck is invalid.

The version of the "fdncheck" command is not the correct one.

Fdncheck terminated abnormally.

"Fdncheck" received a program interrupt from the operating system. The user cannot determine the source of such an error; however, it is not indicative of a problem with either "diskrepair" or the device. "Diskrepair" should be rerun, for the problem may not recur.

Intentional system stop. Reboot system.

If the SIR of the root device must be updated, "diskrepair" kills all tasks running on the system and locks up the system so that no new tasks can begin. It then modifies the SIR. This procedure is necessary to prevent conflicts between the written data and similar data kept in memory. After updating the SIR, "diskrepair" stops the system and prints this error message. The user must reboot the system before proceeding.

No device specified.

The user did not specify a device on the command line.

No such device.

The user specified a nonexistent device on the command line.

Not a block device.

"Diskrepair" can only operate on block devices.

Output directed to device under test.

When testing the structure of a disk, it is impractical to try to redirect the output (the results of the test) to a file on the disk being tested. The user should reexecute "diskrepair" without redirecting the output or redirecting it to a different, mounted device.

Permission denied.

A user who executes "diskrepair" without the 'n' option must have both read and write permission on the specified device. A user who executes "diskrepair" with the 'n' option needs only read permission.

Problems encountered. Diskrepair should be rerun.

"Diskrepair" may encounter more problems than it can fix during one run. For example, it can only handle a certain number of duplicate or out-of-range blocks. If "diskrepair" cannot fix all the errors it encounters, or if it encounters an I/O error but continues operation, it prints this error message when it finishes.

Unknown option: '<char>'

The option specified by <char> is not a valid option to the "diskrepair" command.

Unmount error: <error_num>

"Diskrepair" encountered some problem other than a busy device when it tried to unmount the device. The accompanying error number is the number of the 4404 error that caused the failure. The user should consult the operating system manual for an explanation of the error.

SEE ALSO

blockcheck
fdncheck

fdncheck

Check the integrity of the structure of the file descriptor nodes (fdns) on the specified disk.

SYNTAX

```
/etc/fdncheck <dev_name>
```

DESCRIPTION

The "fdncheck" command checks the integrity of the structure of the file descriptor nodes (fdns) on the specified disk. An fdn contains all the information that the operating system needs to know about a file.

This information includes, but is not limited to, the type of file, the owner of the file, the size of the file, and the addresses of all the blocks that are a part of the file. The "fdncheck" command locates problems such as unreferenced files, directory entries with invalid associated files, and so forth.

This command is primarily intended for use by the "diskrepair" utility, which calls it. It may also be used on its own. However, "fdncheck" can only check the structure of the disk; it cannot repair it. If the output from the command suggests that the structure of the fdns is damaged, the user should execute "diskrepair" on the disk.

The "fdncheck" command should be executed only when no other tasks are active on the system. Otherwise, the results are unpredictable.

Arguments

The name of the device to check. It must be a block device.

EXAMPLES

```
/etc/fdncheck /dev/floppy0
```

This example checks the structure of the fdns on the disk in floppy drive 0.

makdev

Create a special type of file, representing a device.

SYNTAX

/etc/makdev <file_name><dev_type><maj_dev_num><min_dev_num>

DESCRIPTION

The "makdev" command creates a special type of file which represents a device. This type of file allows the user to access the device drivers for the corresponding physical device. Only the "system" user may invoke this command.

Arguments

<file_name>	The name of the file to create. For a block device, the last component of the file name must consist of a string of letters followed by a string of digits. For a character device, the last component of the file name must consist of the same string of letters, followed by the letter 'c', followed by the same string of digits.
<dev_type>	A letter designating whether the device is a block device, (b), or a character device, (c).
<maj_dev_num>	A number which tells the operating system which set of device drivers to use for the specified device.
<min_dev_num>	A number which tells the operating system which physical device to associate with <file_name>.

EXAMPLES

1. /etc/makdev /dev/floppy b 0 0

2. /etc/makdev /dev/floppyc c 3 0

The first example creates a special file named "/dev/floppy", which represents a block device. Currently, all block devices have the same major device number, 0. The first four (beginning with 0) minor device numbers for this major device number designate floppy disk drives 0 through 3. Thus, this command tells the operating system to use the device driver for block devices and to associate the file with the floppy drive.

The second example creates a special file named "/dev/floppyc", which represents the character device associated with the block device "/dev/floppy". The major device number for a character device associated with a floppy disk drive is 3. The first four (beginning with 0) minor device numbers for this major device number designate floppy disk drives 0 through 3. Thus, this command tells the operating system to use the device driver for a character device associated with a floppy disk drive and to associate the file with the floppy drive.

NOTES

- Every disk device requires both a block device and a corresponding character device in order to function properly.

ERROR MESSAGES

'<char>' is not a valid type of device.

The argument <dev_type> must be either 'b', for a block device, or 'c', for a character device.

Error creating "<file_name>": <reason>

The operating system returned an error when "makdev" tried to create the special file <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Invalid major device number: <num>

The number specified as the major device number is invalid.

Invalid minor device number: <num>

The number specified as the minor device number is invalid.

Syntax: /etc/makdev <file_name> <dev_type> <maj_dev_num> <min_dev_num>

The "makdev" command expects exactly four arguments. The command line does not conform to the syntax.

You must be system manager to run "makdev".

Only the "system" user may execute the "makdev" command.

mount

Insert a block device at a node of the directory tree structure.

SYNTAX

/etc/mount <dev_name> <dir_name> [r]

DESCRIPTION

The "mount" command temporarily inserts a block device at a node of the directory tree structure. As long as the device is mounted, any references to <dir_name> actually access the root directory of the device mounted there. Any files in the directory at which the device is mounted are inaccessible while the device is mounted.

Arguments

<dev_name> The name of the device to mount. It must be a block device.

<dir_name> The name of the directory on which to mount the specified device.

Options Available

r Mount the device for reading only. This option must not be preceded by a plus sign. It is useful when trying to salvage data from a damaged disk because it prevents inadvertent writing to the disk, which could make matters worse.

EXAMPLES

1. /etc/mount /dev/floppy /usr2

2. /etc/mount /dev/disk1 /usr2 r

The first example mounts the disk in the floppy drive on the directory "/usr2". References to "/usr2" now access the root directory of that disk.

The second example mounts an accessory hard disk drive, disk1, as "/usr2". Because the 'r' option appears on the command line, no user may write to the disk.

NOTE

When a user's working directory is the root directory of a mounted device, the command "chd .." does not change the working directory.

ERROR MESSAGES

<dev_name>" is not a block device.

The device specified either does not exist or is not a block device. Only block devices may be mounted.

Error mounting "<dev_name>" on "<dir_name>": <reason>

The operating system returned an error when "mount" tried to insert the specified device in the directory tree. This message is followed by an interpretation of the error returned by the operating system.

Only read option allowed for mode.

The only acceptable option is the 'r' option, which must not be preceded by a plus sign.

Syntax: /etc/mount <dev_name> <dir_name> [r]

The "mount" command expects exactly two arguments and, optionally, the single option 'r'. This command indicates that the command line does not conform to the syntax.

SEE ALSO

unmount

unmount

Unmount a previously mounted device from the file system.

SYNTAX

/etc/unmount <dev_name>

DESCRIPTION

The "unmount" command unmounts the specified device from the file system. Once the device is unmounted, the files in the directory on which it was mounted become accessible. Only the system manager may execute this command.

Arguments

dev_name> The name of the device to unmount.

EXAMPLES

/etc/unmount /dev/floppy

This example unmounts the floppy drive.

ERROR MESSAGES

Error unmounting "<dev_name>": <reason>

The operating system returned an error when "unmount" tried to unmount the specified device. This message is followed by an interpretation of the error returned by the operating system.

Syntax: /etc/unmount <dev_name>

The "unmount" command expects exactly one argument. This message indicates that the argument count is wrong.

SEE ALSO

mount

Section 4

4404 ASSEMBLY LANGUAGE PROGRAMMER'S GUIDE

INTRODUCTION

This section, the 4404 Programmer's Guide provides a general introduction to 68000/68010 assembly language programming on the 4404. This section includes a sample 4404 utility program that you can type in and execute.

For detailed information on system calls, see Section 6, System Calls. For detailed information on the 68000/68010 assembler, see Section 6, The Assembler and Linking Loader. System programming in C is described in Section 7, HE 4404 C COMPILER, while programming in other languages is described in the reference manuals for those languages.

SYSTEM CALLS OVERVIEW

The following paragraphs give an overview of assembly language programs on the 4404: how they run, how they perform system function calls, how they handle errors, and what the task environment is like.

HOW 4404 PROGRAMS RUN

Most programs or utilities are run by the user's typing the name of such a program in response to a prompt from the shell. The shell assumes the name which was typed is a file containing an executable binary program. (There are exceptions such as command text files and others, but we will ignore those for now). This binary program is loaded into memory and executed. If desired, this program can obtain parameters from the calling line. When it is finished, the program terminates, passing control back to the shell.

Every program that runs on the system is a task. Many tasks may be active at once, but in reality only one task is running at any given instance. The system switches from task to task so rapidly that the appearance is that all of the tasks are executing concurrently. If you were to freeze the system at some point in time, you would see a single task or program in the cpu's address space. The task may not have all of RAM assigned to it, but it would have the entire address space available. Other tasks may be resident in other memory, but that memory is not mapped into the cpu's space. When the task terminates, its allocated memory is returned to the system, and control is passed to the parent task (the task which created or initiated the terminating task).

SECTION 4
Programmer's Guide

This section discusses how to write the program which the shell can load and execute, how this program can communicate with the user, system, other tasks, etc, and how to terminate the program's execution.

INTRODUCTION TO SYSTEM CALLS

When a user's program communicates with the user, a disk file, another task, or anything else in the system, It uses calls to the operating system. The operating system is essentially another task, is always available, that has built in routines to perform a multitude of system oriented functions. These functions include reading files, writing files, seeking to file locations, setting permissions, creating pipes, reporting id's, creating tasks, terminating tasks, mounting devices, reporting the time, and so on. A user program can execute these functions by making a call to the system with a proper function code and input parameters. The technique of making the call in the assembler code is the "sys" instruction recognized by the assembler.

THE "SYS" INSTRUCTION

The assembler has a built-in instruction to make system calls. It is the "sys" instruction and has the following format:

```
sys <function>,<parameter1>,<parameter2>,...
```

The only required portion of the operand is the <function>, which is simply a numeric code for the desired function. The parameters required depend on the particular function. There may be no parameters or as many as four. The function code is a 16-bit value; while parameters are always 32-bit values. Many system functions also require certain values or parameters to be in one or more of the 68010 cpu's registers before executing a "sys" instruction. When some parameters are required in registers, it is the programmer's responsibility to see that the proper values are loaded before calling on the system.

When the "sys" instruction has completed execution, control generally passes to the next instruction in the program. In some cases, the system function returns one or more values to the calling program, by placing the values in selected cpu registers. In some cases the returned value or values will be placed at a location specified as one of the input parameters.

Section 7, System Calls, describes the operating system functions. Along with the description, the necessary parameters and returned values are specified. For example, look at the "read" system call in that section. Under the USAGE heading you will see the following:

```
<file descriptor in D0>
sys read,buffer,count
<bytes read in D0>
```

This shows that before executing the read function call, you must ensure that the desired "file descriptor" must be loaded into the 68010's D0 register. In addition to the read function code itself, you must supply a buffer address (32-bit address of a buffer to read into) and a count (32-bit count of how many characters to read). After the read function has been executed, the actual number of bytes read will be returned in the 68010's D0 register.

All user-accessible 68010 registers except for the D0, A0, and CCR registers are left intact across system calls. The contents of the D0, A0, and CCR registers upon return from a system call vary depending on the particular call.

The actual system function code numbers in the "sysdef" file located in the "/ lib" directory. This file is provided on disk so that you can include those definitions in your program by including the "sysdef" file in your source via a "lib sysdef" instruction.

Briefly, the "sys" function works by generating a software interrupt. When this interrupt occurs, the handling routine maps the calling task out of the cpu's address space and maps the operating system code in. This system code then performs the requested function. It obtains the function number and parameters from the code directly following the software interrupt itself. When the system function has completed, the operating system is mapped out, and the task is mapped back in to continue with its instructions.

SYSTEM CALL EXAMPLE

Before cluttering things up, let's try a sample program that includes a system function call. The simplest program would be one which did nothing at all: as soon as it is initiated, it will immediately terminate. Thus the only system function we will need to call is the "term" function. The description of "term" in Section 7, System Calls shows that there are no parameters required on the "sys" instruction itself (besides the function code), but that you must put a status value in the D0 register before performing the call. If there are no errors this status should be zero. Thus you can write an extremely simple program that looks like the following:

```
          lib       sysdef
start     move.l    #0,d0      Get status in D0
          sys       term       Terminate task
          end       start
```

The first line includes the definitions of all system function codes so that we can specify the term function as a symbol ("term") and not have to type in the particular number for that function. The second line puts the status in D0, as required by the "term" function, and line 3 terminates the program. In the case of the "term" function, control is not returned to the calling program after execution of the call. Of course, that is the reason for the function; it terminates the current task (the task which made the call) and returns control to that task's parent. Notice that the program's end statement includes the symbol "start". This tells the assembler the beginning location for execution and also induces the assembler to make the resulting code executable by setting the permission bits.

Let's assume you call the source file "nothing.txt" and assemble it with the following command:

```
++ asm nothing.txt +ls +o=nothing.r
++ load nothing.r +o=nothing
```

The result would be a binary file which when executed by the command:

```
++ nothing
```

would load, run, and immediately return to the shell. This is, of course, a meaningless example, but it does show the rudimentary steps in writing, assembling, and executing a 4404 assembly language program.

INDIRECT SYSTEM CALLS

In order to use the "sys" instruction directly, all the parameters must be defined at assembly time. When parameters are not known at assembly time (because they will be determined or changed during the execution of the program), you must use indirect system calls. There are two types of indirect system calls -- "ind" and "indx" -- and they are themselves system functions called with the normal sys instruction. They permit the programmer to tell the system that the parameters do not actually follow the software interrupt, but instead are placed at some other specified location in memory. This memory location, specified by the programmer, can be in an area of memory containing data and not program code.

The first of these indirect system call functions is called "ind". Its format is:

```
sys ind,label
```

The "label" is the address of the memory locations that will contain the actual desired function code and parameters. Thus, when this function is executed, the system goes to location "label" and picks up the desired function code and any necessary parameters. The system executes that function and returns control to the statement following the "sys ind,label" instruction.

To illustrate, let's assume a program that needs to read from a file, but does not know how many characters to read until it is executing. Somewhere in the first part of the executing program, the number of characters to be read is determined and stored in a label called "rcount." The indirect function call is used:

```
            ...
            ...
            move.l    rcount,iread+6      Get count to read
            move.l    fd,d0               Get file descriptor
            sys       ind,iread           Do indirect read call
            ...
            ...
iread       dc.w      read                READ function code
            dc.l      buffer              Read buffer location
            dc.l      0                   Read count (unknown)
buffer      ds.b      $4000     Space for read buffer
```

(At this point we're not concerned with details of how the read really works or what the file descriptor is, we simply want to show how the indirect system call is made.)

The second form of indirect system call is the "indx" function, and is very similar to the "ind" function. The difference is that the call to "ind" includes a parameter ("label") that points to the parameters in memory; with the "indx" function the pointer to the parameters in memory is in the A0 register. To see how this works, we can modify the above sample by changing the instruction "sys ind,iread" to:

```
        lea     iread,a0        Get address of parameter
        sys     indx            Do indirect read call
```

An obvious use of indx is to push the parameters onto the system stack and point A0 to it, thereby eliminating the need for the parameter buffer in memory. For example:

```
        ...
        ...
        move.l  rcount,-(a7)    Setcount to read
        move.l  #buffer,-(a7)   Set buffer address
        move.w  #read,-(a7)     Set read function code
         move.l fd,d0            Get file descriptor
        move.l  a7,a0           Point to parameters on
                                stack
        sys     indx            Do indirect read call
        lea     10(a7),a7       Clean parameters off
       stack
        ...
        ...
buffer  ds.b    $4000           Space for read buffer
        ...
        ...
```

Note the importance of the order in which the parameters are pushed onto the stack. Also note the "lea 10(a7),a7" instruction following the function call. It removes the parameters which were pushed onto the stack so that the stack is where it was before the system call section.

SYSTEM ERRORS

Upon completion, system calls return to the calling program with an error flag. This flag is the carry bit in the condition code register. If the bit is zero on return, it implies that no error occurred. If the bit is set (a one), then an error has occurred and the D0 register contains an error number. The assembler supports two special mnemonics for testing the error status on return from a system call: "bes" for "branch if error set" and "bec" for "branch if error cleared." These are equivalent to the standard mnemonics "bcs" and "bcc."

Section 7, System Calls, contains a list of the error numbers and their meanings. There is also a file of equates called "/lib/syserrors" which assign standard labels to the error numbers. These can be used in a program by simply including the file with a "lib syserrors" instruction. Note that the operating system does not report errors directly to the user. Error numbers are returned from system calls and it is entirely up to the user's program to report such errors or handle them as required by the specific application.

THE TASK ENVIRONMENT

A "task" is a single program which has complete use of the cpu's directly-accessible address space. It can call on functions in the operating system, but is essentially a single, stand-alone program. Each time a program is run, a new task is generated and the program becomes that task. Whenever that executing task performs some I/O or system call that will require it to wait, the task is mapped out so that another waiting active task may be mapped in and executed. If the executing task does not perform any type of system call which would cause it to be mapped out, it will eventually run into a time-slice interrupt which will force the task out so that other tasks can get some execution time.

In this manner, multiple tasks can be run at what seems like the same time. To assist in keeping track of all the active tasks, the operating system assigns a unique "task id" number to each task. This is a 15 bit unsigned value that can be used to uniquely identify a particular task. The "gtid" system call allows a task or program to obtain this task id if desired.

ADDRESS SPACE

The addresses which can be generated by a program make up what is known as the logical address space. Under hardware memory management, these logical addresses are not presented directly to the system memory. Instead, they are routed through the hardware memory manager, which translates the logical addresses into physical addresses. Memory management allows programs which reside at a particular logical address to actually load into system memory at a different physical address. The total range of physical addresses makes up the physical address space.

Although it would be possible to pass the addresses generated by the program directly to the system memory, the use of a hardware memory manager provides several benefits. First, and perhaps foremost, it prevents one task from reading from or writing to the memory allocated to another task. In addition, it allows multiple tasks to reside in physical memory without the need for each task to reside in a different area in the logical address space. Thus, all programs can be written to execute at the same fixed logical address. No matter where those programs are loaded into physical memory when they are executed, the memory management unit converts the logical addresses used by the program to the proper physical addresses.

The 4404's logical address space is divided into three sections: text, data, and stack. The program itself resides in the text section. This section cannot be written to during execution of the program. The data section contains any data used by the program. It can be both read from and written to during execution. The system stack is located in the stack section. The memory management unit allocates a certain amount of memory to each section when the task is initiated. The amount of memory assigned to each section is determined by the size of the task and its needs. It is also possible, as we shall see later, for a task to add more memory to the data or stack section during execution.

ARGUMENTS

It is often desirable to pass arguments or parameters to a program when you begin its execution. The "exec" system call provides this ability. "exec" is the call which is used to begin execution of a program or binary file.

Arguments are passed to a program by leaving them on the system stack. When the program is initiated, the system stack pointer (A7) is left pointing at some unknown location in the stack page. Any arguments passed to the program are found in a special format just above where the stack pointer points. The arguments themselves are simply strings of characters which the program must know how to use. In order to easily find these strings, the system provides a list of pointers to the beginning of the strings. In addition, the system provides a count of how many arguments have been passed. This argument information is laid out as follows:

1. The stack pointer is pointing to the argument count. It is a 4 byte value and should always be greater than zero.

2. Just above the argument count (higher addresses in memory) is the list of pointers to the argument strings. These pointers are 32 bit addresses of the actual strings.

3. At the end of the pointer list are four bytes of zero to signify the end of the list. (A null pointer.)

4. The actual string arguments begin above the zero bytes. Each argument string is the string of characters that make up the argument followed by a zero byte.

Let's look at an example. Assume that whoever started our task passed us three parameters: the name of our program, a file name, and an option which starts with a plus sign. The system always passes the name of the program or command being executed as the first argument (argument number 0). Assume the program name is "pile", the specified file name is "data2", and the option is "+b". Our argument count will be three. Let us arbitrarily say the system stack pointer is at $FFFFFDE0. We should see the following data on the stack:

item	location	contents
arg 2 terminator	$FFFFFE01	$00
argument 2	$FFFFFDFF	'+b'
arg 1 terminator	$FFFFFDFE	$00
argument 1	$FFFFFDF9	'data2'
arg 0 terminator	$FFFFFDF8	$00
argument 0	$FFFFFDF4	'pile'
arg list terminator	$FFFFFDF0	$00000000
pointer to arg 2	$FFFFFDEC	$FFFFFDFF
pointer to arg 1	$FFFFFDE8	$FFFFFDF9
pointer to arg 0	$FFFFFDE4	$FFFFFDF4
argument count	$FFFFFDE0	$00000003

Thus if we want to get the second argument (argument number 1), we read the pointer stored at the stack pointer + 8. This value, $FFFFFDF9, is the pointer to the argument string itself. At the location to which the pointer points is the string of characters "data2" followed by a zero byte.

In general, the programs or utilities that a system programmer writes will be initiated by the shell. Specifically, they will be started when the user types the name of that program in response to the shell's prompt. The shell starts the program by performing an "exec" system call. The arguments that the shell sets up for the exec (which are those passed to the program) are the arguments that are typed on the shell command line after the program name. By convention, the shell sets argument 0 to be the command or program name itself. The arguments after the program name are then numbered sequentially beginning with one. If our "pile" program above were an executable binary file, the arguments described above would result from a shell command line that looked like this:

```
++ pile data2 +b
```

The shell performs pattern-matching before passing the arguments to the command. For example, consider the command:

```
++list file*
```

The shell does not pass "file*" as an argument to list, but rather searches the directory for all filenames that match and passes them all as individual arguments. Thus, the list program would see four arguments:

```
argument 0 -> list
argument 1 -> file1
argument 2 -> file2
argument 3 -> filename
```

(Recall that argument number zero is always the name of the program or command being executed.)

INITIATING AND TERMINATING TASKS

In a multi-tasking environment, one task can spawn or start a new task. There must, of course, also be means for terminating tasks and for the parent of a terminating task to be informed of that termination. The following discussion covers these techniques.

TERMINATING A TASK

Tasks or programs are terminated with the "term" system call. When this function is executed, the task is halted and its memory is relinquished to the system. Before calling the "term" function, the programmer is required to place an error status value in the D0 register. When the task terminates, this value is passed back to the task's parent. If there is no error on termination, this error status should be zero to indicate a clean termination. If the task terminates due to a system error such as an I/O error, the error value returned by that system call should be used as the error status for the term function. If the task terminates due to an error defined by the program (for example, the program expects an argument but none was supplied), the recommended value to return is $000000FF. By convention the parent task would recognize this as a user defined error. The parent would know some error had occurred that caused the program to terminate, but would not be able to determine the exact nature of the error. A user-defined error should not return a termination status of greater than $000000FF.

THE "WAIT" SYSTEM CALL

The "wait" system function is issued by a task when it wants to wait for one of the children tasks it has spawned to terminate. It is through the wait command that the parent task receives the termination status from its child. "wait" has the following syntax:

```
sys wait
```

When the system call returns, the termination status is in the A0 register and the terminated task's id is in the D0 register.

If there are no children tasks when a wait call is issued, an error will be returned. If a child task is still running when the parent issues a wait, the parent will be put to sleep until the child task has terminated. If a child task terminates before its parent has issued a wait, the system will save the child's task id and termination status until the parent does issue a wait. If several children tasks have been spawned, the parent must issue a wait call for each one individually.

The termination status is a two-byte value that is returned in the lower half of the A0 register. The lower byte (bits 0-7 of A0) is the low-order byte of the status value passed by the "term" system call. If this byte is non-zero, some sort of error condition caused termination. Under normal termination conditions, the higher byte of the termination status (bits 8-15 of A0) is zero. If non-zero, the task was terminated by some system interrupt, and the least significant seven bits of this byte contain the interrupt number. If the most significant bit of this byte is set, a core dump was produced as a result of the termination. (Interrupt numbers and core dumps will be described later.)

THE "EXEC" SYSTEM CALL

At times, a user-written program may wish to load and execute a program by itself without going back to the shell. The tool used to load and execute another program or binary file is the "exec" system function. That is the very function which the shell uses when it loads and executes a program. (Remember the shell itself is just another program.)

The program which makes the exec call is thrown away and the new program (a binary file) is loaded into memory and executed. The same task id number is retained. If the exec is successful (i.e. no errors such as the file not existing), there will be no return to the calling program. The calling program is thrown away, making it impossible to return. If, however, there is an error in attempting to perform the "exec" function, the system will not load the new program but will return an error status to the calling program which is still intact. Thus a properly written program will follow any "sys exec" call with error handling code.

The "exec" call requires two arguments: a pointer to the name of the file to be executed and a pointer to a list of arguments to be supplied to the new program. "exec's" format is:

```
sys exec,fname,arglist
```

The "fname" is the pointer to the filename. This filename is a string of appropriate characters located somewhere in memory and terminated by a zero byte. The "arglist" is the pointer to a list of argument pointers. In other words, "arglist" is an address at which we will find a list of pointers. This list of pointers is consecutive 4-byte addresses or pointers to the actual argument strings. The list is terminated by four bytes of zero (which could be considered a pointer to zero). Each pointer in the list is the address of the actual argument string also terminated by a zero byte. When the exec function is complete, the new program will have these arguments available in the exact format previously described.

Let's try an example of the use of exec. As you know the "ls" command can be run by typing the name and possible arguments on the shell command line. The shell actually starts execution of ls by performing an exec. As an exercise, let's write our own program that executes the ls command automatically, always providing an argument of "+ba". This will provide a long directory with file sizes specified in bytes and which includes all files. We will not specify any specific directory, so our command will always perform the directory command on the current directory. The filename to exec should be "/bin/ls," and there will be two arguments, "ls" and "+ba". We supply "ls" as argument zero because by convention argument number 0 is the command name. Our program looks like this:

```
          lib   sysdef

start     sys   exec,filen,args

* This point is reached only if the exec fails.  There
* would normally be error handling code here, but to keep
* things simple, we will just terminate if an error.
* Note the D0 register already has the error from exec.

          sys       term

* strings and data

filen     fcc       '/bin/ls',0
arg0      fcc       'ls',0
args      fcc       '+ba',0
args      ds.l      arg0,arg1,0
          end       start
```

If we called this utility "ls-ba", after assembling we could execute it by typing "ls-ba" as a command to the shell. Our program would be loaded and executed by the shell, and it would in turn load and execute the ls command with a "+ba" option. Thus typing "ls-ba" would produce the same results as typing "ls +ba".

THE "FORK" AND "VFORK" SYSTEM CALLS

The "fork" and "vfork" system calls are used to spawn a new task, and are the only way to create new tasks. They create a new task which is almost identical to the old (the old task is still around). This new task has the same memory and stack allocation, same code in the memory space, same open files, pointers, etc. Thus, immediately after a fork, there are essentially two identical tasks or programs running on the system. Usually you want the new task to do something different, so in most cases the new task will immediately perform an "exec" call to load some program from disk and execute it. This is the technique used by the shell to start background jobs. When the shell sees a command ending with an ampersand ("&"), instead of directly doing an "exec" it does a fork to create a second shell. Now the newly created shell will do an "exec" of the desired command, while the old shell is still around to accept further commands.

The syntax of either fork command is simply:

```
sys fork  -- or
sys vfork
```

The tricky part of the fork call is in how the two almost-identical tasks know which is which. If the two tasks have the same code, how can the new one do an exec while the old one does not? The answer is in the return from a fork call. After the fork operation, execution will resume in each of the two programs. The difference is in where that execution resumes. In the new task, execution resumes in the instruction immediately following the fork system call. The old task resumes execution at a point two bytes past the system call. In this manner, the same program can be run in two tasks via a fork and yet do different things after the fork. Since the new task resumes directly after the fork call and the old task resumes two bytes after the fork call, it is obvious that the first instruction in the new task must be a short branch instruction (which requires only two bytes). Note that the new task's id is made available to the old task by supplying the id in the DO register upon return from the fork. If an error occurs when attempting a fork, the new task will not be created, and an error status will be returned to the old task (still two bytes past the fork system call).

The following section of code will help illustrate the fork:

```
        ...
        ...
sys     fork    spawn new task
```

```
* new task begins execution here

          bra.s     newtsk    branch to code for new task

* old task resumes execution here

          bes       frkerr    check and branch if error
          move.l    d0,d1     save new task's id
prwait    sys       wait      wait for child task
          cmp.l     d0,d1     right one?
          bne.s     prwait    wait some more if not
          ...                 continue code for old task
          ...
          sys       term
newtsk    sys       exec,name,args      new task probably
                                        does exec
          bra       excerr    branch if error in exec
          ...
          ...
```

In this example, the "wait" system call at "prwait" makes the old task wait for the new one (it's child) to finish before continuing. Note that the "wait" system call returns the terminated task's id in the D0 register.

4404 FILE HANDLING

This topic describes the manipulation of files, terminals, directories, printers, and other devices on the 4404.

GENERAL FILE DEFINITIONS

Before delving into the actual manipulation of files on the 4404, we need to define and describe some of their characteristics.

Device Independent I/O

Under the 4404 operating system, anything outside the program's memory to which the program can write or from which it can read, is treated the same. A file on disk, a terminal, a pipe, and a printer spooler are treated the same way. This concept, termed "device independent I/O" means you can develop a program that sends its output to a terminal, and that same program, without change, will also be able to output to a disk file, printer spooler, pipe, or any other device on the system. This feature lends a great amount of versatility to the system and makes program development and updating much smoother.

This device independence is made possible by device driver routines -- the system routines that take care of the specifics of the device for which they are written, creating a standard interface to the device. There is a routine to open the device and one to close it. These permit the system to do anything necessary to prepare the device for reading and writing and to finalize anything necessary when all I/O is complete. The two most important device driver routines are the "read" and "write" routines, which permit the caller to read or write data from the device.

File Descriptors

A file descriptor informs the system which file to operate on. (We use the term "file", but because of device independence, the file descriptor can refer to a disk file, terminal, pipe, or any other device). The file descriptor is a four-byte numeric representation of a specific file or device. This number is assigned to the file by the system when that file is opened or created. The operating system then keeps track of the file descriptors and the files to which they are assigned. In this way, the user supplys a number instead of an entire file name each time the file is to be referenced.

For example, the "read" system call requires a file descriptor value in the D0 register before making the call. In general use, we would have saved the file descriptor number of the file we wish to read when it was opened. Now, to do the read, we need only load the D0 register with that number.

File descriptor numbers begin with 0 and extend up to the maximum possible number of open files on the system. This maximum will vary depending on the system configuration, but generally will be around 12-25.

Standard Input and Output

When the shell begins execution of a task, it automatically assigns input and output files to that task. Generally the input file is the user's keyboard, and the output file is the user's display. In fact, when a task begins execution, it can always count on three input/output files being already opened, assigned a file descriptor, and ready for reading or writing: "standard input," "standard output," and "standard error output." Standard input is an open file ready for reading and is always assigned a file descriptor of 0. Generally the standard input file is the 4404 keyboard. Standard output is an open file ready for writing to and is always assigned a file descriptor of 1. Generally the standard output file is the 4404 display. Standard error output is an open file ready for writing to and is always assigned a

file descriptor of 2. This output file is reserved for reporting error messages. It is almost always the 4404 display.

Because these standard input and output files are already opened and assigned a file descriptor, the user program does not have to perform any "open" or "create" calls in order to perform I/O activities on them. As soon as a task begins running, it can perform a read with a file descriptor of 0 (standard input) or write with a file descriptor of 1 or 2 (standard output and error output).

Standard input and output can be "redirected" without any change to the program. In other words, a program which outputs some message to the user's terminal can also output the message to a disk file without any modifications. This I/O redirection is accomplished from the shell by use of the "<" and ">" operators (redirected input and output, respectively). If the shell desires, it can provide a standard input or output file to the program which is different from the user's terminal. The user program need not be concerned with what the standard input or output is pointing to. Because of device independence and the fact that the program knows that the file or device (whatever it may be) has been previously opened, the program simply performs the I/O and doesn't care where it's going.

OPENING, CLOSING, AND CREATING FILES

Before a file or device can be read from or written to, it must be opened. When a program has completed all its input and output to a file, it should generally close that file. A user program may also need the ability to create new files on the system. This topic addresses those operations in detail.

The "open" System Call

The format of an "open" system call is:

```
sys open,fname,mode
```

The "fname" is a pointer to a zero-terminated string containing the name of the file to be opened. The "mode" is a number (0, 1, or 2) which sets the read/write mode. If 0, the file is opened for reading only. If 1, the file is opened for writing only. If 2, the file is opened for both reading and writing.

On return from the open call, register D0 will contain the 4-byte file descriptor number assigned to that file. All future references to the file will be made via this file descriptor.

An error will be returned from this call if the file to be opened does not exist, if the task opening the file does not have proper permissions, if too many files are already opened, or if the directory path leading to the file cannot be searched.

The "close" System Call

When a task terminates, the operating system automatically closes any files that remain open. It is wise, however, to manually close files within a program whenever possible. There are two reasons for doing so. First, since the system has a finite number of files which may be open at one time, closing a file will free up a slot in which another file may be opened. Second, in case of a system crash, you will be better off having closed any files which no longer require I/O. The "close" system call is performed by loading register D0 with the file descriptor of the file you wish to close, then performing a "sys close".

The "create" System Call

The "create" system call is used to create disk files. Other system calls are used to create directories, pipes, devices, etc. The format of create is:

```
sys create,fname,perm
```

Once again, "fname" is a pointer to a zero-terminated string containing the name of the file to create. The file will be created in the default directory unless a directory is explicitly specified in the file name. The "perm" is a value which permits the user to set the desired permissions on the new file. (Refer to Section 7, System Calls for details of setting these permissions.)

Note that if the file already exists in the specified directory, it will be truncated to zero length (all existing data deleted). In addition, the original permissions will be retained regardless of the "perm" value supplied to the create call. In other words if the file "fname" already exists, the "perm" parameter on the create call will be ignored.

If the file does not exist, permission setting will be subject to any default permission settings the file owner has previously specified. The "perm" parameter in the "create" call allows you to deny permissions which the default permissions grant, but does not let you grant permissions that the default permissions deny. You can think of this as a logical AND of the "perm" parameter and the default permission byte.

Every task has associated with it a default permissions byte. If that task attempts to create any new tasks, the new tasks are created with at least those default permissions. As we saw above, additional permissions may be denied by the "perm" value specified to a "create" call. Additionally, the new task is started with the same default permission byte (for creating more tasks) as it's parent. In normal use, a user may set the default permissions in his copy of the shell upon first logging on. If the default permissions are not changed by the user or any task he runs, any files the user creates will have those default permissions. (Note that the user can change default permissions with the "dperm" command and for a task to change its own default permissions with the "defacc" system call.

READING AND WRITING

Perhaps the most heavily used system calls are "read" and "write." It is by these functions that a program communicates with the user, disk files, printers, other tasks, and anything else in the outside world. Reading and writing permits great versatility in how files are accessed. For example, with a disk file, the user can begin at any particular point in the file (right down to a specific character) and read or write as many characters as desired from that point. This makes both sequential and random access of the files quite simple.

The "read" and "write" system calls assume a "file position pointer" has already been set. This is a pointer which the system maintains to show the current position for reading and writing in a file. The discussion on "seeking," later in this section, shows how it can be set. The only parameters required, then, are the file descriptor to specify which file, the count of characters to be read or written, and a memory buffer address to read into or write from.

The "read" System Call

To execute a "read" call, the programmer must first load register D0 with the file descriptor number. Then he or she makes the "read" call with the following syntax:

```
sys read,buffer,count
```

The "buffer" parameter is an address in the user program's memory. It specifies where the data read from the file should be placed in memory. The "count" is the maximum number of characters the programmer wants the system to read. We say maximum because, depending on the situation, the system may not actually read as many characters as requested. Upon return from the read system call, register D0 contains the number of bytes that was actually read.

When dealing with a regular disk file, the system will always read "count" bytes if possible. There are only two reasons that the system would read less than that number from a regular disk file: a physical I/O error occurs, or the specified count forces the system to attempt to read past the end of the file. For example, if a file has only 120 characters and a "read" call is issued with a "count" parameter of 256, the read will take place and return with no error, but will show that only 120 characters were actually read. After this call the file position pointer will be left pointing at the end of the file. Any subsequent read call will return with no error, but with the number of bytes read equal to zero. This is in fact how a user program should detect an "end of file" condition: a return from a read system call with no error but with the actual number of characters read being zero.

Reading and writing to terminals is handled with the same system calls as when reading and writing disk files. There is a difference in the result of a read call, however, in that if the file being read is a terminal, only one line will be returned at most. By one line we mean all the characters typed since the last carriage return, terminated by a carriage return. Thus, even though we execute a call with a desired "count" of 1024 characters to be read, if the user at the terminal types the letters "halt" followed by a carriage return, the read call would return with an actual-bytes-read count of only five. If the user has not typed anything when the call is issued, the calling program must wait until something is typed.

As with regular disk files, it is possible to detect an "end of file" condition from a terminal by performing a "read" and receiving no error and no characters. An "end of file" condition from a terminal is produced by typing a Control-D. Note that the Control-D character itself is not actually passed on to the operating system, only the "end of file" condition.

As an example of the use of the read call, let's examine a section of code that attempts to read 1024 bytes of data, placing them in a buffer called "buffer". We assume the file has already been opened for reading and the file descriptor is stored at "fdsave".

```
          ...
          ...
          move.l    fdsave,d0     get file descriptor
          sys       read,buffer,1024     read 1024 bytes into
                                          buffer
          bes.l     rderr          branch if error
          tst.l     d0             end-of-file-condition?
          beq.l     endof          special handling if so
          add.l     #buffer,d0      point to end of data
          move.l    d0,bufend       save buffer end pointer
          ...
          ...
buffer    ds.b      1024
          ...
          ...
```

Upon return from the "read" system call, we first check for a returned error status. If an error occurred, we assume the program handles it properly at "rderr". If no error, we check for an "end of file" condition. Recall that an "end of file" condition is recognized by a program as zero characters read when there was no error. If we are at the end of the file, the program jumps to "endof," where we again assume that such a condition is properly handled. If we did not receive an error and were not at the end of the file, our program calculates a pointer to one past the last byte read into the buffer and stores that pointer at "bufend". Normally this pointer should be "buffer+1024", but if the read call returned less than 1024 bytes it would be lower.

The "write" System Call

The "write" function is executed by
first loading register D0 with the file descriptor number and then issuing the "write" call:

```
sys write,buffer,count
```

The "buffer" parameter is the address of the location in the user program's memory where the data to be written is located. The "count" is the number of characters to be written to the file. Upon return from the "write" system call, the D0 register will contain the actual byte count written (if there is no error). It is not necessary to compare this value to the requested count to be written because if there was no error, you can be sure the entire write function took place properly.

Let's look at a complete program to send the message "Hello there!" to the standard output file. If there is an error in writing to that file, we will then send the message "Error writing standard output." to the standard error output file. (Recall that the standard output is assigned file descriptor number 1 and standard error output is assigned file descriptor number 2.)

```
          lib     sysdef          include system definitions

* start of main program

sayhi     move.l  #1,d0           write to standard. output
          sys     write,hello,hlng  send message
          bec.s   done            exit if no error
          move.l  d0,-(a7)        else, save error number
          move.l  #2,d0           write to std. error output
          sys     write,erm,elng  send error message
          move.l  (a7)+,d0        restore error number
          bra.s   done2
done      move.l  #0,d0
done2     sys     term            terminate program

* strings

hello     fcc     'Hello there!',$d,0
hlng      equ     *-hello          compute length of string
erm       fcc     'Error writing standard output.',$d,0
elng      equ     *-erm           compute length of string

          end     sayhi           give starting address
```

There is no "open" system call because we know that the standard output and standard error output files are already opened and ready for writing when the program begins execution. Note the convenient method of providing the count of characters to be written. Also note that we did not look for an error after the system call to write to the standard error output. We really have no good recourse if an error does occur while reporting an error, so we simply terminate.

Efficiency in Reading and Writing

There are several things a system programmer can do to achieve efficient reading and writing of files on the 4404. The first and most obvious of these is to read or write as much of a disk file as possible with a single call. There is much less system overhead in executing one call to read 4096 characters than in executing 32 calls to read 128 characters each. The most efficient reads and writes are those made in multiples of 512 bytes. This is, of course, due to the fact that the 4404 disk block size is 512 bytes. Due to the way memory mapping works, additional efficiency can be gained by placing all read and write buffers on 512 byte address boundaries in memory.

By all means do not perform single character I/O with system calls for each character. If single-character I/O is required, the user program should handle the necessary buffering so that system calls are made only on a buffer full of characters.

SEEKING

For each open disk file, the operating system maintains a pointer that indicates the current position for reading or writing in that file. This pointer can point to any place in the file, right down to any specific character position. The user does not have direct access to this pointer, but use the "seek" system call position it to any desired spot in a file. The format of the seek call is:

```
sys seek,offset,type
```

Before making a system call to "seek", the user must load the desired file descriptor in register D0. Seeks are done on a relative basis. That is, a seek amount is supplied to the call and the seek is to be that amount relative to some reference point. (This reference point is the "type" parameter shown above.)

There are three possible reference points: the beginning of the file, the current position in the file, and the end of the file. The "type" value should be as follows:

type	starting position or reference point
0	beginning of the file
1	current position in file
2	end of the file

The argument "offset" is a four-byte 2's complement offset that represents the amount of offset to be added to the reference point to find the new position in the file. A positive number indicates forward in the file; a negative number indicates backward into the file. On return from the "seek" call, the new current position is left in register D0. This is the current position relative to the start of the file. To find the current position in a file, you could use a system call of "sys seek,0,1", finding the result in D0.

As an example, let's construct a simple random access routine. Assume we have a data file with fixed-length records of 256 characters per record. We know we will never have more than 32000 records in our file, so the record number can be represented in 16 bits. We want to write a subroutine that will read the record specified by the record number in register a0 and leave the data at the location specified by the A0 register. The basic procedure will be to find the starting position of the desired record in the file by multiplying the record number by the record size of 256. Then we seek to that position and read 256 bytes. Our routine looks like this:

```
          ...
          ...
getrec    move.l    a0,iread+2      save address for read
          ext.l     d0              make record number long
          lsl.l     #8,d0           record*256 is offset

* seek to record

          move.l    d0,iseek+2      set seek address parameter
          move.l    fd,d0           assume file descriptor at fd
          sys       ind,iseek       indirect call to seek
          bes.l     skerr           branch if error

* file pointer positioned, now read record

          move.l    fd,d0           get file descriptor
          sys       ind,iread       indirect call to read
          bes.l     rderr           branch if error
          rts                       all finished
          ...
          ...  iseek     dc.w       seek            seek function
code
          dc.l      0               seek address (unknown)
          dc.l      0               type 0: position from begin
iread     dc.w      read            read function code
          dc.l      0               buffer location (unknown)
          dc.l      256             character count to read
          ...
          ...
```

Notice that we used indirect calls to "seek" and "read," because at assembly time we do not know what address we will need to seek nor where in memory to place the data we read. By using indirect calls, we can set aside an area of memory (at "iseek" and "iread") where these values can be stored as the program executes.

FILE STATUS INFORMATION

The "status" and "ofstat" calls are used to obtain information about each file or device. "ofstat" is used to obtain information about a previously opened file while "status" obtains information from an unopened file. The format for ofstat is:

```
<file descriptor in D0>
sys ofstat,buffer
```

The user must load register D0 with the file descriptor of the previously opened file.

The format for status is:

```
sys status,fname,buffer
```

With "status", the file is specified by providing the "fname" parameter, which is a pointer to a zero-terminated string containing the desired file name. In both commands the "buffer" parameter is a pointer to a buffer in memory or an area of memory into which the information about the file can be placed. This buffer must be at least 22 bytes long. When the "status" or "ofstat" call is completed, this buffer will contain all the information available for the file in the format described below.

Assuming the buffer begins at some location called "buf", the information in the buffer is as follows:

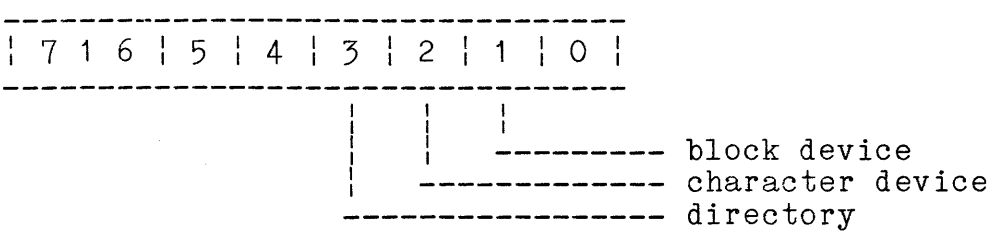
Name	Location	Field Size	Information in Field
st_dev	buf	2	device number
st_fdn	buf+2	2	fdn number st_fil
buf+4	1		spare (for word alignment)
st_mod	buf+5	1	file mode
st_prm	buf+6	1	permission bits
st_cnt	buf+7	1	link count
st_own	buf+8	2	file owner's user id
st_siz	buf+10	4	file size in bytes
st_mtm	buf+14	4	time of last file modification
st_spr	buf+18	4	reserved for future use

The device number is a number assigned to the device on which the file resides. The fdn number is the number of the "file descriptor node" associated with the file. The file descriptor node is a block of information about the file and where it resides on the disk. It is from the fdn that "status" and "ofstat" obtain their information.

The link count is the number of directory entries that are linked to the fdn or actual file. More information on linking can be found later in this section in the discussion titled "Directories and Linking." The file owner's user id is a two-byte id that was assigned to the user by the system manager when the user was given a user name. The file size in bytes is the exact number of characters in the file. The time of last modification is the internal representation of the last time the file was written to.

The file mode and permission bytes each hold several bits of information. This is done by assigning single bits within the file mode to particular file types and within the permission byte to the various possible permission types. The state of the particular bit (0 or 1) indicates which type of file mode or whether permission is given or denied. The file mode looks like this:

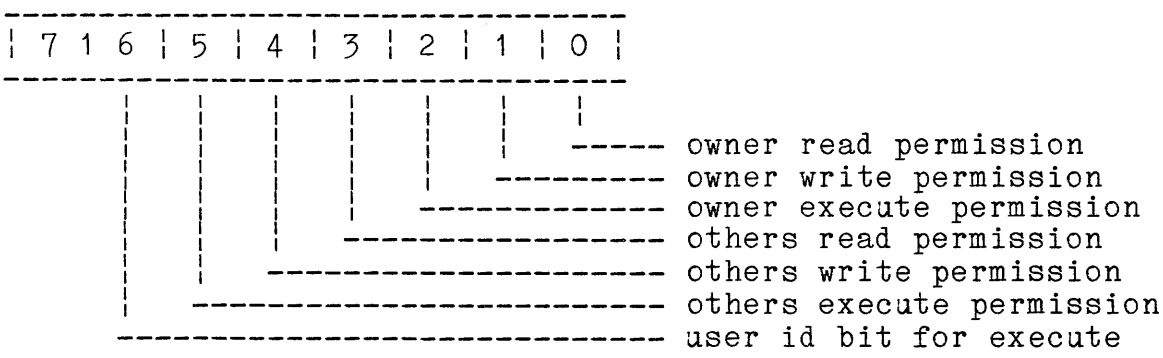
file mode (st_mod):



Notice that only three bits are used in this byte. Only one of the three bits should be set at a time and it indicates the file type. If the file is a regular disk file, none of the bits will be set. A block device is a device such as a disk drive which handles data in 512 byte blocks. A character device is one such as the sound device (/dev/sound) that handles data single character at a time.

The permissions byte shows what permissions are granted or denied for the file. Its format is as follows:

permissions (st_prm):



In this byte, any or all of the permission bits may be set at one time. If a bit is set, that type of permission is granted. If cleared, permission is denied.

The "user id" permission bit requires further clarification. If this bit is set, it gives the user of a file the same permissions as the owner while that file is executing. As an example of the usefulness of this feature, consider a user, "joe", who has a database program which manipulates a large data file. Now "joe" does not want anybody on the system to be able to directly read or write his data file, so he denies read and write permissions on that file to others. (Of course, he grants read and write permissions for himself.) Even though he does not want anyone to be able to read and write his data file directly, "joe" would like for other users to be able to run his database program, which manipulates the data file. All he need do is set the "user id" permission bit in his database program. With the "user id" bit set, anyone who runs the database program has the same permissions as "joe," which allows them to manipulate the data file while running the database program. As soon as the database program is terminated, however, the other user no longer has the permissions of "joe," the owner.

Another example of the use of the "user id" bit can be seen in the "crdir" or "create directory" command. A directory is a special type of file, and the only way to create a directory is with the "crtsd" system call. That call may only be executed by the system manager. Without the "user id" bit, the only person who could use the "crdir" command (which contains a "crtsd" system call) would be the system manager. The "crdir" program has the "user id" bit set, however, so that anyone who runs it temporarily has the same permissions as the owner. The owner of "crdir" is the system manager; thus any user can create a directory.

DIRECTORIES AND LINKING

A directory entry is nothing more than the name of a file and a single pointer to the file descriptor node (fdn) for the file. This fdn is a small unit on the disk; it contains various information about a particular file. There is one and only one fdn on a disk for each file which resides on the same disk. It is possible, however, to have more than one directory entry point to the same fdn. Two different users could have an entry in their own directory which pointed to the same fdn and therefore the same file. This feature is called a "link" and you can see it is possible to have many "links" to the same file.

A long directory listing (ls +l) shows the number of directory entries which point to or are linked to each file. This is always "1" or greater; if it ever goes to zero no one is linked to the file and it will be deleted. In fact when you "remove" a file, the command merely removes that name from the directory. This decrements the link count in the associated fdn. If that count is still non-zero, someone else is linked to the file and it is not deleted from the disk. If the count does go to zero, no one else is linked to the file and it is deleted.

An example of linking can be seen in every directory on a 4404 disk. Recall that there are two entries, "." and ".." in each directory. (They don't appear in a "ls" listing unless you use the "+a" option.) The "." entry represents the directory in which that entry is found; ".." represents the parent directory of the directory in which it is found. Thus typing "." as a directory name is equivalent to typing the entire path name for the current directory. Typing ".." is equivalent to typing the path name for the parent directory of the current directory. These directory entries are not separate files, but are links to the current directory file and the parent of the current directory. That is why you see a link count of more than one for every directory on the system.

The "link" and "unlink" system calls allow the programmer to link to files and unlink from files, respectively. The "link" function is quite straightforward: one specifies a pointer to the name of the file to be linked to, and a pointer to the new name that will be put into the directory. The "unlink" call is equally straightforward: the programmer simply provides a pointer to the filename or directory entry to be unlinked. This "unlink" call is the method of deleting files, the "remove" command calls on the "unlink" function to perform the file deletion. Note that a file is not deleted by an "unlink" call unless the call removes the last link to the file.

If a file is open when an "unlink" call is made, the link is removed, but the file will not be deleted or closed by the operation. The user can still read or write to the file as long as it is left open. The 4404 operating system waits until the file is actually closed and then checks the link count to see if it should be deleted from the disk. This creates interesting possibilities for a program. A file can be opened and then immediately unlinked. As long as the program leaves that file open, it can read from it or write to it. When the program is finished with the file, it has only to close it. If no one else is linked to the file, it will be immediately deleted.

OTHER SYSTEM FUNCTIONS

This discussion describes several features and functions available to the system programmer that are somewhat specialized. Specific calling formats and parameters will not always be given; for this refer to Section 7, System Calls.

THE "BREAK" FUNCTION

Earlier, we learned that when a task is started, it is allocated text, data, and stack memory according to the program size. It is possible for a running task to change the amount of memory allocated to it's data or stack spaces. It is also possible to relinquish allocated memory back to the system, that is to deallocate data or stack memory. The means of performing this dynamic memory or stack allocation and deallocation are the "break" and "stack" commands. An address is supplied to break and the system attempts to allocate memory to be sure there is RAM up through the specified address. Memory is allocated in sections, so depending on the address specified there may be some memory beyond the address. If an address is specified which falls below the amount of program memory already allocated, that memory is relinquished or returned back to the system.

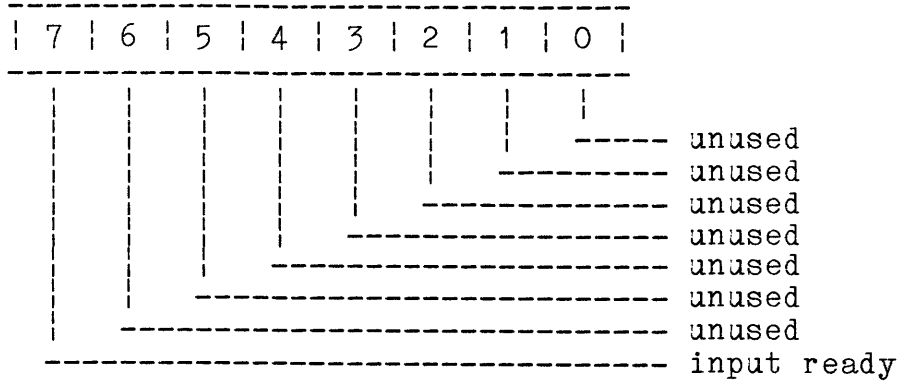
THE "TTYSET" AND "TTYGET" FUNCTIONS

The 4404's "ttyset" and "ttyget" functions provide a way to alter and examine several configuration parameters of devices. (The communications port and console devices differ slightly in format.) These parameters include such things as the line-cancel character, the backspace character, adjustable delay after carriage returns, mapping of upper to lower case, tab expansion, etc. The configuration of all these parameters is represented in six bytes of data. These six bytes can be read with the "ttyget" system call to examine the current configurations, or can be set with the "ttyset" system call to alter the current configuration. A six-byte buffer must be established in memory to hold the desired configurations for "ttyset" or to receive the current configuration information for "ttyget." If we assume this buffer begins at "ttbuf", the data has the following format:

Name	Location	Contents
tt_flg	ttbuf	Flag byte
tt_dly	ttbuf+1	(reserved)
tt_cnc	ttbuf+2	Line cancel character (default is Ctrl-U)
tt_bks	ttbuf+3	Backspace character (default is Ctrl-H)
tt_spd	ttbuf+4	Terminal speed
tt_spr	ttbuf+5	Stop output byte

The Terminal Speed byte presently implements only one bit. It is the high order bit (bit 7) and, if set, indicates that the terminal has input characters waiting for the program. This bit is meaningful only when read, i.e. the input-ready condition cannot be set via this bit and "ttyset." The byte looks like this:

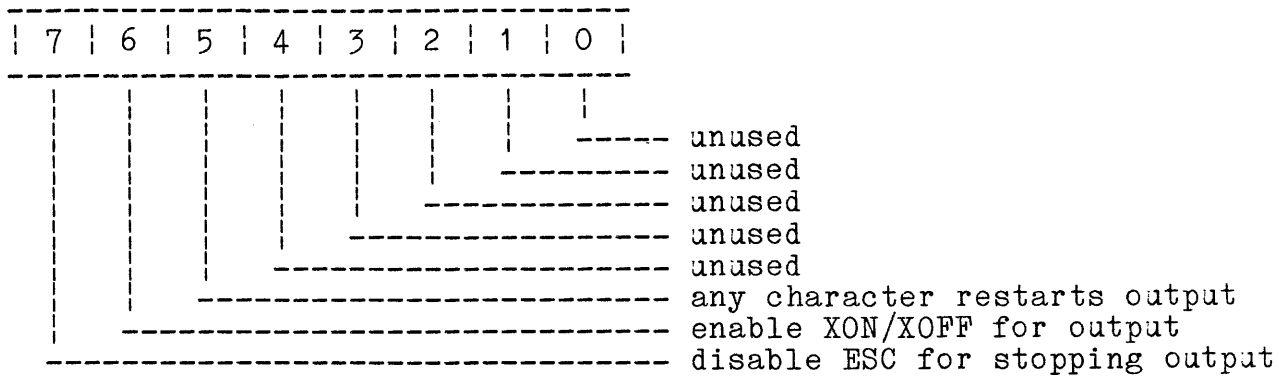
Terminal Speed byte (tt_spd):



Under normal input operations, the "Input Ready" bit is not set until an entire line has been input and terminated by a carriage return. There are special input modes which can be established, however, where the "Input Ready" bit will be set as soon as a single character is input. These are the "raw I/O mode" and the "single character input mode", and are described later in this section.

The Stop Output byte contains bits which control the stopping and starting of output to terminals. There are two methods by which a user can stop and start output to a terminal: the escape key and XON/XOFF processing. The escape key method permits a user to type an escape character (hex 1B) to stop output. A subsequent escape character restarts the output. The XON/XOFF method permits a user to type an XOFF character (hex 13) to stop output and a subsequent XON character (hex 11) to restart it. Many terminals produce XON and XOFF characters automatically to prevent the computer from sending too many characters to the terminal at once. The escape and XON/XOFF mechanisms can be independently enabled or disabled by setting or clearing the proper bits in the "tt_spr" byte. The byte looks like this:

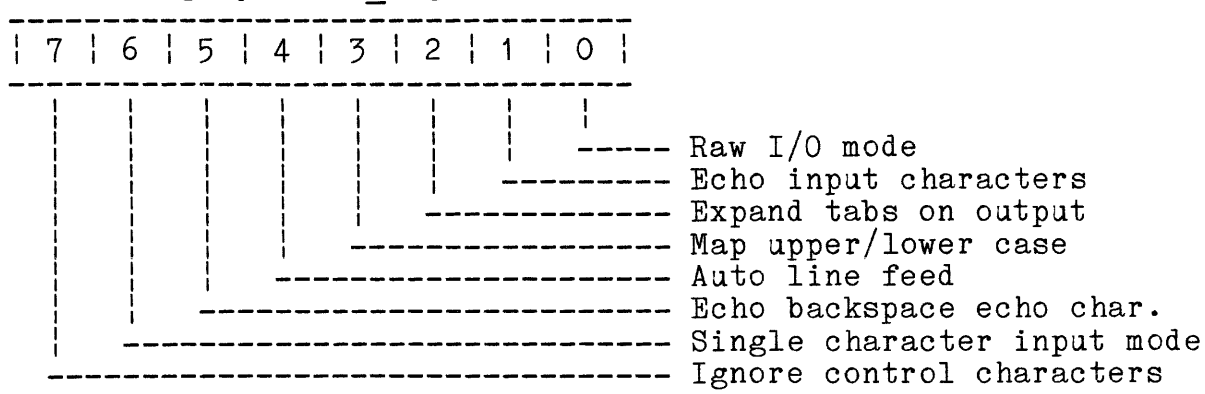
Stop Output byte (tt_spr):



When set, "Any Character Restarts Output" bit instructs the terminal drivers to restart the output if it has been stopped by either an escape or XOFF.

The eight bits of the Flag byte represent eight different modes of operation for the terminal. When set, they imply that the indicated mode is in operation. The format is as follows:

Flag byte (tt_flg):



The following paragraphs describe each of these modes.

Raw I/O Mode

In "raw mode", the terminal drivers effectively do no special processing of the input or output characters. Each and every character typed on the terminal is directly input, including backspace characters, line cancel characters, tab characters, Ctrl-C characters, and so on. Similarly, every character output to the terminal is output directly: no tab expansion is performed, no line feeds are appended to carriage returns, etc. In addition, the parity bit is not stripped on either input or output.

In "raw mode," the executing program has complete control of every character input or output and the program must perform any special processing itself. Under raw mode a "read" system call will not have to wait for an entire line to be input before it can read characters. If there is a single character available, the "read" call will return with just that character. It is still possible for a single "read" call to read more than one character, but only if the characters have already been typed into the input buffer before the call is made.

Echo Input Characters

If this mode is enabled, each character typed on the terminal will be echoed to the display device. In such a case, the terminal should be operating in full-duplex. An example of this mode occurs when a user logs in and is asked for his password. The login program writes the "Password:" message and then turns the "echo input characters" bit off while the password is entered. In that way the password is not echoed to the screen. This mode is on by default.

Expand Tabs on Output

If the terminal does not have hardware tab expansion, this bit can be set to allow the terminal driver software to automatically expand tabs on output. Tab stops are assumed to be at 8 column intervals. In other words, if this bit is on, then each time a horizontal tab character ($09) is output, the system will space over to the next column which is a multiple of 8 (unless it is already at such a column). This mode is on by default.

Map Upper/Lower Case

The 4404 assumes that the terminal has upper and lower case capability and that the user will type most commands and input in lower case. It is possible, however, to use an "upper case only" terminal by instructing the terminal drivers to map all typed input characters from upper to lower case and to map all output characters from lower to upper case. This is done by turning on the "Map Upper/Lower Case" bit in the ttyset flag byte. When this mode is on, then mapping is done in both directions. By default, this mode is off (assumes lower case capability). It is automatically turned on, however, if a user logs in with upper case characters for his name. In this way an "upper case only" terminal can be connected to the 4404 without special considerations.

Auto Line Feed

When this mode is on, the terminal drivers will automatically output a line feed ($0A) after each carriage return is output. This mode is on by default.

Echo Backspace Echo Character

If this mode is on and the backspace character is defined to be a Ctrl-H ($08), the terminal drivers will echo the Ctrl-H, then output a space, and then output another Ctrl-H. This will erase the incorrect character for terminals which do not do so automatically. This mode is on by default.

Single Character Input Mode

"Single Character Input Mode" allows a program to input one character at a time without having to wait for a carriage return. When not in the single character input mode, a call to read a single character would have to wait until an entire line terminated by a carriage return had been typed before it would have access to a single character within the line. If single character input mode is on, the program can read a character as soon as it has been typed. Note that it is still possible to read multiple characters while in the single character input mode, if they are available. While in the single character input mode, the parity bit is stripped off of input characters, but only Ctrl-C, Ctrl-D, and Ctrl- are treated as special characters. In other words, tabs, backspaces, and line cancels are ignored and should be processed by the user's program if desired. This mode is off by default.

Ignore Control Characters

When this mode is on, the system will ignores all control characters except for the following:

- Carriage Return
- Horizontal Tab
- Ctrl-C
- Ctrl-D
- Ctrl-
- Backspace Character
  (if defined to be a control character)
- Line Cancel Character
  (if defined to be a control character)

Those control characters which are ignored will still be echoed if the echo input characters mode is also on. This mode is off by default.

PIPES

A pipe is a mechanism that permits a task to communicate with a child task.

A pipe allows communication in one direction only; it allows one task to send information to another, but not to receive. If a pair of tasks need two-way communication, two pipes must be established; one to send from the first task to the second and one to send from the second task to the first. Once the pipe is established, the first task sends information to the second by using the "write" system call, just as it would in writing to any other device. The second task receives information from the first by using the "read" system call. The file descriptor numbers for these write and read operations are provided by the system when the pipe is created.

The pipe mechanism works sort of like a holding tank with a valve on the input and output lines. If the tank is not full, the writing task can pump data into it even though the reading task has the output valve closed (is not actively reading). Likewise, if the tank is not empty, the reading task can drain information out of it even though the writing task has the input valve closed (is not currently writing). If the tank is full, the writing task is forced to wait until the reading task has emptied it before being permitted to pump in more data. If the tank is empty, the reading task must wait until the writing task has pumped in some data. This "holding tank" is a 4K disk buffer. There is a buffer for each pipe, but none show up in any directory. These pipe buffers are placed on the disk unit which has been configured as the pipe device.

The following section of code establishes a pipe between a task (A) and its child task (B). First, Task A calls "crpipe" to create the pipe. Next, we immediately fork to create Task B, and then set up the file descriptors so that we will be writing from task A to task B. The code would look something like this:

```
            ...
            ...
            sys       crpipe          create pipe system call
            bes.l     piperr          branch if error
            move.l    d0,rdfd         save read file descriptor
            move.l    a0,wrtfd        save write file descriptor
            sys       fork            fork to spawn task B
            bra.s     child           new task B here
            bes.l     frkerr          task A checks for error
            move.l    d0,tskBid       save task id of child
            move.l    rdfd,d0         pipe read file descriptor
            sys       close           close read (A only writes)
            move.l    wrtfd,pipefd    save pipe write file descriptor
 * now Task A can write to pipe using pipefd
            ...
            ...
            sys       term            end of task A


* code for Task B


child       move.l    wrtfd,d0        pipe write file descriptor
            sys       close           close write (B only reads)
            move.l    rdfd,pipefd     save pipe read file descriptor
* now Task B can read from pipe using pipefd
            ...
            ...
```

Notice that each task closes the portion of the pipe that it cannot use. As previously stated, a pipe allows data to be transmitted in only one direction. After performing the fork, both tasks have open read and write pipe files. Now it is assumed that the writing task will eventually close the write pipe file, and the reading task will eventually close the read pipe file. However, we must be sure that the writing task closes the read file and the reading task closes the write file. In fact, these files should be closed as soon as possible, before any reads or writes to the pipe are performed.

PROGRAM INTERRUPTS

Program interrupts provide a way to interrupt tasks under software control. One program or task can send a program interrupt to another task. This permits timing and synchronization among the tasks in the system. It also gives the programmer the ability to terminate tasks prematurely under software control.

Sending and Catching Program Interrupts

Here is an example of how a program sends an interrupt.

```
...
...
move.l    #327,d0          get task number in D0
sys       spint,SIGQUIT    send quit interrupt
bes.l     error
...
...
```

Assuming the effective user id of the task executing the above code matches that of task number 327 or that the above task is owned by the system manager, a "quit" interrupt will be sent to task 327. (We will define the quit interrupt and other interrupts in a moment.) Notice the system call used to send program interrupts is "spint". It is also possible for a program to send an interrupt to all tasks associated with the terminal which executed the program. Consult the "spint" description in Section 7, System Calls for details.

The "cpint" (for "catch program interrupt") provides a way for a task to "catch" or intercept a program interrupt when it is received. The task may then permit the interrupt to complete its default action (usually task termination), may ignore the interrupt completely, or may take some special user-defined action.

In effect, "cpint" permits the user to set up an interrupt vector address, so that if a program interrupt is received, control is vectored to that address. The programmer may place a routine at that address which handles the interrupt in some special way. Two addresses, $000000 and $000001, are special. If the address specified for the caught interrupt is $000000, the default action of the interrupt will be allowed to occur, much as if the interrupt had not been caught at all. If the address specified is $000001, the interrupt will be ignored, much as if the interrupt had not even been sent. Note that no code is actually placed at these addresses. The "cpint" function recognizes them as special values and performs the indicated interrupt handling without ever jumping to or using them as real addresses. Any other address supplied to "cpint" is assumed to be a valid program memory address, and control is passed to that location. There, the programmer places the desired interrupt handling routine; this routine must be exited with an RTR instruction, so that control is resumed at the same point in the program where the interrupt occurred.

Once a program interrupt has been caught and processed, the system resets itself back to the default condition, and interrupts are no longer intercepted. Therefore, to continue catching program interrupts, the programmer must issue a new "cpint" call after each interrupt is processed.

Table 4-1 shows the program interrupts that are available on the 4404.

Table 4-1

4404 PROGRAM INTERRUPTS

Name	Number	Description	Comments
SIGHUP	1	hangup interrupt	
SIGINT	2	keyboard interrupt	
SIGQUIT	3	quit interrupt	produces core dump
SIGEMT	4	EMT $AXXX emulation int.	produces core dump
SIGKILL	5	task kill interrupt	can't be caught/ignored
SIGPIPE	6	write broken pipe int.	
SIGBUS	7	bus fault	
SIGALRM	10	alarm interrupt	
SIGTERM	11	task termination interrupt	
SIGTRAPV	12	TRAPV instruction	produces core dump
SIGCHK	13	CHK instruction	produces core dump
SIGEMT2	14	EMT $FXXX emulation int.	produces core dump
SIGTRAP1	15	TRAP #1 instruction	produces core dump
SIGTRAP2	16	TRAP #2 instruction	produces core dump
SIGTRAP3	17	TRAP #3 instruction	produces core dump

SIGTRAP4	18	TRAP #4 instruction	produces core dump
SIGTRAP5	19	TRAP #5 instruction	produces core dump
SIGTRAP6	20	TRAP #6 instruction	produces core dump
SIGILL	22	illegal instruction	produces core dump
SIGDIV	23	divide by zero	produces core dump
SIGPRIV	24	privilege violation	produces core dump
SIGADDR	25	address error	produces core dump
SIGDEAD	26	dead child task interrupt	ignored by default
SIGWRIT	27	write to read-only memory	produces core dump
SIGBND	29	segmentation violation	produces core dump
SIGUSR1	30	user-defined interrupt #1	
SIGUSR2	31	user-defined interrupt #2	
SIGUSR3	32	user-defined interrupt #3	

If not caught or ignored, all of these program interrupts (except SIGDEAD) by default cause termination of the task to which they are sent. As listed above, some also produce a "core dump". A "core dump" is a disk file which contains a mirror image of the contents of memory. Each byte in the program and stack space are written to a disk file immediately after receipt of the interrupt. This file can be examined to determine the state of memory at the time the interrupt was received. This is often useful for diagnostic purposes.

Many of the interrupts are initiated by 68010 exception processing. The cause of those interrupts can be understood by studying the documentation of the 68010 microprocessor. Certain interrupts in the list are not directly initiated by the 68010 and need further definition.

1. Hangup Interrupt: Generated by the operating system when a terminal driver loses the carrier that it had previously established for modem operation. This interrupt causes the user associated with the terminal to be automatically logged off. Certain programs (such as the editor and BASIC) intercept this interrupt and take proper actions to save current files before logging off.

2. Keyboard Interrupt: Generated by typing a Ctrl-C on the terminal. This interrupt terminates the foreground task of the associated terminal.

3. Quit Interrupt: Generated by typing a Ctrl-Backslash on the terminal. This interrupt is just like the Keyboard Interrupt except that it additionally produces a core dump.

4. EMT $AXXX Emulation Interrupt: Generated by the 68010 when an instruction with the pattern 1010 in bits 15 through 12 is encountered.

5. Task Kill Interrupt: Always kills the task to which it is sent. A task may not catch or ignore this interrupt.

6. Write Broken Pipe Interrupt: Generated when a pipe between two tasks is broken. This occurs when the reader is closed and the writer attempts further writing.

10. Alarm Interrupt: Generated by the "alarm" system call after the specified number of seconds. Unless caught or ignored, this interrupt will terminate the task.

11. Task Termination Interrupt: This interrupt is the normal means of interrupting and terminating a task. Unlike the Task Kill Interrupt, the Task Termination Interrupt may be caught or ignored.

14. EMT $FXXX Emulation Interrupt: Generated by the 68010 when an instruction with the pattern 1111 in bits 15 through 12 is encountered.

26. Dead Child Task Interrupt: When a task terminates, it sends an interrupt to its parent task, informing the parent that the child has terminated. This interrupt is ignored by default--it must be explicitly caught by the parent in order to function. At the time of this writing, the Dead Child Task Interrupt is not implemented.

27. Write to Read-Only Memory: An attempt was made to write to a section of memory which has been reserved as Read-Only by the memory management system.

29. Segmentation Violation: An attempt was made to access memory which is outside the address space allotted to a task.

30-32. User-Defined Interrupts: These interrupts are additional interrupts which a user program or set of programs may issue and catch for whatever purpose they wish.

On return from a "cpint" call, register D0 contains an address. This address is the address which the system was using on receipt of program interrupts. In other words, it is the address which was provided in the previous "cpint" call. This old address can be used to tell what kind of action a program was taking on receipt of program interrupts before the current "cpint" call. For example, assume we have a program that is ignoring quit interrupts. If we now issue the instruction:

```
sys cpint,SIGQUIT,0
```

(which says to take the default action on receipt of a quit interrupt) we would find "1" returned in the D0 register. That 1 is the address which was previously being used, and we know that an address of 1 says to ignore the interrupt.

Knowing what type of program interrupt action is currently being taken can be very useful in the case where one task starts another. If one task is ignoring some particular interrupt and that task starts some new task running, the new task should usually also ignore the interrupt. Assume we Program A starts Program B by doing a "fork" and "exec". Also assume Program B normally wishes to catch keyboard interrupts (Ctrl-Cs) and process them in a special way. Program B should be written to first check how Program A was handling keyboard interrupts. If Program A was not intercepting keyboard interrupts or was catching them, Program B may go ahead and catch them and process them as desired. If, however, Program A was ignoring keyboard interrupts, then Program B should also ignore them. The code for Program B to handle all this properly would be:

```
          ...
          ...
          sys       cpint,SIGINT,1        Start by ignoring
          cmp.l     #1,d0                 Was program A ignoring?
          beq       contin                If so, then so should we
          sys       cpint,SIGINT,handle   If not, catch it
contin    ...
          ...
```

Note that by ignoring the keyboard interrupt while checking what Program A was doing, we avoid a potential chance for a keyboard interrupt to come through and be improperly handled.

As an example of program interrupt catching, let's examine a portion of code that would put a program to sleep for 30 seconds. The technique will be to send an alarm interrupt with the "alarm" system call, then put the task to sleep with the "stop" system call. In order to catch the "alarm" interrupt and continue properly in our program, we will use the "cpint" system call.

```
          ...
          ...
          sys       cpint,SIGALRM,wake    catch alarm & goto wake
          move.l    #30,d0                delay 30 seconds
          sys       alarm
          sys       stop                  wait for alarm interrupt
          ...                             continue with program
          ...
          ...
wake      rtr                             do nothing with interrupt
          ...
          ...
```

The "cpint" system call tells the task to catch any alarm interrupts and handle them as specified by the code at "wake". In this example the code at "wake" does absolutely nothing but return. That is because when the alarm is received we want to simply continue execution of the program where we left off (just after the "stop" system call).

Interrupted System Calls

Most system calls cannot be interrupted by a program interrupt. That is, once a system call is executing, it will finish regardless of whether a program interrupt is pending. Once that system call is completed, the user's program will then see any waiting program interrupt. There are a few calls, however, which may be terminated by a program interrupt. In particular, those system calls which may be interrupted are "read" and "write" (if the device being read or written is a slow device such as a terminal or printer) and the "stop" and "wait" calls. A "read" or "write" call to a fast device, such as a disk file, will never be terminated by a program interrupt.

If a program interrupt does get through to one of the system calls, the following action takes place. First, the system call is immediately terminated, and control is passed to the program interrupt handling code if the interrupt is being caught. Then, when the interrupt handling code is complete, control is passed to the instruction immediately following the interrupted system call **and an error status is returned.** This error status is accompanied by an "EINTR" error (number 27). In this way, the program which made the system call can detect that it was interrupted and re-issue the system call if desired.

As an example, consider a program which prompts the user for a line of data from the terminal. If a program interrupt is sent to that program while a "read" system call is getting the data from the terminal, that call may be prematurely terminated; i.e. not all the data may be returned. Once the program interrupt handling code was complete, our program would continue right after the "read" call, but would show an "EINTR" error. Our program may choose to treat the EINTR error like any other and terminate with an error message. An alternative , however, would be to recognize that it was an EINTR error and loop back in our code to re-issue the prompt and the "read" system call to input the data again.

LOCKING AND UNLOCKING RECORDS

The "lrec" and "urec" system calls provide a record locking mechanism that prevents more than one task attempting to access a file at one time. A program or task can "lock" a record of data until such time as it is ready to "unlock" or release it for others to use. While that record is locked, no other task would be able to access it.

The operating system maintains a table showing what records are locked in the system. These records may be of any length, as specified by the task which performs the lock. Note that a single task may lock only one record in a file. However, other tasks can lock other records in that same file, and a single task can lock a record in more than one file at a time.

When a task issues an "lrec" call to lock some record within a file, the system first checks the locked record table to see if the calling task already has a record locked in this file. If so, any such record is unlocked before the new record lock can be made. Next, the system checks to see if the record to be locked is available or if some other task may have previously locked some portion of it. If available for locking, the system makes an entry in the locked record table and returns to the calling task. If the desired record overlaps some portion of an already locked record, the system returns with an ELOCK error. At this point, the calling program could take some appropriate action.

There are three ways for a task to unlock a record. The first is through use of the "urec" system call, which unlocks whatever record may have been locked by the calling task for the specified file. The second is by closing a file. Upon closing, any records locked by the task that opened the file are automatically unlocked. The third is by locking another record in the same file; this will automatically unlock any record which is currently locked.

Having said this, we must back up and tell you that "locking" a record does not really prevent another task from accessing it. Any program that wishes to can still read or write the data which some other program has locked in a record. In order for locking to provide the desired results, all programs must take upon themselves the responsibility of avoiding reading or writing to a locked record. This may be accomplished by attempting to lock records before reading or writing them. If the record is available, no error is returned, and we can go ahead with the read or write. If an error is returned (ELOCK error), we know that someone else already has the record locked and we should take some other action. One possibility is to put our task to sleep for a few seconds (with the "alarm" and "stop" system calls), and then try locking the record again. Proper use of the lock and unlock calls will yield the same result as if locking actually did prevent another task from reading or writing. Note that locking and unlocking will not be necessary in all cases, only in those where a data file is shared and conflicts can occur.

SHARED TEXT PROGRAMS

The 4404 operating system lets you separate an assembly language program into two sections, a "text" segment for nonchanging memory or memory which will only be read, and a "data" segment for memory which can be changed by writing into it. When a task runs this program, a section of memory will be assigned to each segment. If a second task runs the program at the same time, the system will recognize the fact that it already has a copy of the text segment in memory and will only load the data segment into memory for the second task. The system will then map the same memory that contains the text segment for the first task into the address space for the second task when it runs. For more details on how to produce a shared text type program, refer to Section 5, The Assembler and Linking Loader.

GENERAL PROGRAMMING PRACTICES

This discussion covers several general programming practices that are recommended when writing assembly language programs to run on the 4404.

STARTING LOCATIONS

Assembly language programs should not have specific origin addresses. Rather, the load addresses for the text and data sections of a program (as well as the stack established by the system) should be specified at load time. These addresses can be explicitly specified to the loader, but should generally assume the default values found in the file "/lib/ std_env". This file contains the proper addresses for the hardware memory manager and is automatically read by the linking-loader.

STACK CONSIDERATIONS

When a program begins execution, it is assigned a portion of memory to contain the program stack. The cpu's system stack pointer (register A7) is left pointing to some location within this memory. The user's program should not write into locations in memory higher than this initial stack pointer location. The passed parameters which lie directly above the stack pointer (higher in memory) may be read, but nothing should be written above the initial stack pointer location.

HARDWARE INTERRUPTS AND TRAPS

In general, a user program need not perform any hardware interrupt or trap handling. Some traps can be handled in the same fashion as program interrupts by using the "cpint" system call.

DELAYS

To maintain system efficiency, a user's program should not contain delay routines which tie up the processor for long periods of time. Because of task switching, a delay loop does not provide accurate timing delays anyway. The preferred method is to use the "alarm" system call followed by a "stop" system call. The program must also then use the "cpint" system call to catch the "alarm" interrupt and continue with the desired code.

SYSTEM "LIB" FILES PROVIDED

Several system library files are provided for the convenience of the assembly language programmer. Located in the "/lib" directory, these files contain definitions for several system related calls, tables, buffers, etc. The programmer may include these definitions in his programs by simply using the "lib" instruction in the 68010 assembler. These files include:

sysdef	System call definitions
sysdisplay	System display and event definitions
syserrors	System error definitions
sysints	Program interrupt definitions
sysstat	Status and ofstat buffer layout
systim	Time and ttime buffer layouts
systty	Ttyget and ttyset buffer layout

An additional file is provided for use by the linking-loader. It is called by the linking loader and should not be included in an assembler program.

std_env	Standard environment for linking-loader

GENERATING UNIQUE FILENAMES

Often, it is necessary for a program to generate a filename. A typical example is when a program wishes to create a scratch file of some sort. In a single-task environment, the program could just use some name defined at assembly time. In a multi-task environment, however, more caution is required. If the program which generates the filename is run as more than one task (background/foreground for example) there may well be conflicts since each copy of the running program would be attempting to create and manipulate the same file. The proper technique to avoid this problem is to have the program include the current task id as part of the filename. Since each executing copy of the program has a different task id, they will each generate different filenames. Use the "gtid" system call to obtain the task id number, then convert it to ASCII and include it as part of the filename.

DEBUGGING

Assembly language debugging on the 4404 is accomplished via the "debug" command. This command provides tools such as memory dumps, breakpointing, and single-stepping. Refer to Section 2, User Commands and Utilities, for documentation on the "debug" utility.

PROGRAMMING EXAMPLE

The following sample utility demonstrates several of the calls and techniques in writing assembly language utilities on the 4404. This utility reads a file (or list of files) and strips out all control characters except for carriage returns ($0d) and horizontal tabs ($09). The syntax of the command line is as follows:

```
strip [file] ...
```

The square brackets indicate that the file name specification is optional. If no filename is supplied, "strip will read the standard input. The three periods ("...") indicate that it is possible to supply more than one file name. In such a case, strip will read all the files in order and write the stripped output to the standard output.

Our basic task, then, is to read either a list of files or the standard input, strip the necessary control characters, and write the result to the standard output device. In order to handle any size file(s), we shall read and write the data into a buffer. We know that for efficiency, the buffer should be an even multiple of 512 bytes, but how big a multiple? The code to implement this utility will obviously be quite small, such that the program and the buffer could easily fit in 4K of memory. Since this utility will probably not be frequently used, we decided to limit the program memory utilization to only 4K. We will make the read/write buffer as large as possible within that 4K space, while keeping it a multiple of 512 bytes.

The first step, after titling and describing the program, is to include the system definitions with the "lib" instruction on line 17. Next we actually begin the code section of our program with the "text" statement in line 23. In line 27 we load the "a6" register with a pointer to the list of filename arguments. The list is null if no filename was specified. Notice that we skip eight bytes, four containing the argument count and four containing argument 0 which is the name of the command itself.

Lines 28 through 31 check to see if a file or files were specified on the command line. If so, the argument count (what the system stack is pointing to) will be greater than 1 because argument 0 (the command name) counts as one. If the argument count is 1, no file was specified, so we must read the standard input. The file descriptor for standard input is 0, so that value is saved in "ifd" and we jump ahead to process that input. If a file was specified, we enter a loop to read through all specified files.

In line 35 we obtain the pointer to the next file in the list and store it at "opname". If that pointer is zero (a null pointer), we have reached the end of the list, and we jump off to the exit code at "done." If it is non-zero, it must be the address of a filename string. Lines 40 through 42 open that file for read and save the file descriptor in "ifd". Note that the open is done via an indirect system call. This is necessary because when the program is written, we do not know what filename to specify in an open call. The pointer to the name of the file to be opened is only discovered as we run the program. When we stored the filename pointer at "opname" in line 35, we were actually storing the filename pointer in the parameter list for the upcoming indirect open system call.

In line 46 we call a subroutine named "strip" to read through the file whose descriptor is in "ifd," strip out the control characters, and write the result to standard output. Line 47 branches back to the top of the loop to look for another possible input file.

The "strip" subroutine is where the control characters are actually stripped. In lines 67 through 69 we read "BUFSIZ" characters into memory at "buffer." Lines 73 and 74 check for end-of-file. If we were at the end of the file, we jump to "strip9" and exit the subroutine. If not, we go on to lines 80 through 91, where the control characters are stripped from the buffer. Note that after the control characters are stripped, the resulting data is left in the same buffer. Because some characters may have been stripped out, the location of the end of the data in the buffer may be lower than before the stripping.

After the stripping, we fall into lines 96 through 101, where the stripped data is written out to standard output. Lines 96 and 97 calculate the number of characters to write. It is equal to the difference between the pointer to the end of the data in the buffer and the pointer to the beginning of the buffer. The result is stored in the parameters for an indirect write call. In line 98 we obtain the file descriptor for the standard output file. Lines 99 and 100 carry out the indirect write system call. In 101 we jump back to the beginning of the subroutine to read in another buffer of data.

Lines 113 through 134 contain the error handling code. If an error occurs, we simply write an appropriate message to the standard error output (file descriptor 2). The important thing to note about this code is that we save the error status so that it may be passed on to the "term" system call.

Lines 144 through 158 contain temporary storage and buffers. First are the parameter lists for the indirect open and write calls mentioned earlier. Line 153 reserves storage space for the current input file descriptor. Lines 155 through 158 reserve the read/write buffer. The buffer starts on a 512 byte boundary and the end of the buffer is the end of the 4K memory page. Recall that read/write efficiency is gained not only by a buffer size which is a multiple of 512 bytes, but also by beginning the buffer on a 512 byte boundary. Line 157 establishes the buffer size by calculating the difference between the end of the 4K page ($1000) and the beginning of the buffer. The "end" statement on line 161 specifies the utility starting address in its operand field.

SAMPLE "STRIP" UTILITY

```
1  ********************************************************************
2  *
3  * Sample "strip" Utility
4  *
5  * Copyright (c) 1984 by
6  * Technical Systems Consultants, Inc.
7  *
8  * Utility to strip all meaningless control characters from
9  * input file and write stripped version to standard output.
10 * Accepts list of input files or defaults to standard input.
11 * For the purpose of this utility, "meaningless control
12 * characters" are all characters with and ASCII value between
13 * $00 and $1F inclusive except carriage return ($0D) and
14 * horizontal tab ($09).
15 ********************************************************************
16
17           lib     sysdef        read system definitions
18
19 ******************************************
20 * start of main program
21 ******************************************
22
23           text                  begin text segment
24
25 * start by seeing if any input files were specified
26
27  start    lea     8(a7),a6      set arg ptr past count & arg0
28           cmp.1   #1,(a7)       file specified only if argcnt >1
29           bhi.s   main2         branch if filenames present
30           move.1  #0,ifd        else use standard input
31           bra.s   main4         go process std. input
32
33 * check to see if any more files specified
34
35  main2   move.1   (a6)+,opname  get next argument in list
36          beq.s    done          branch if no more args
37
38 * open specified file for read
39
40          sys      ind,iopen     do indirect open call
41          bes.s    opnerr        branch if error
42          move.1   d0,ifd        save input file descriptor
43
44  * strip control characters from this file
45
46  main4   bsr.s    strip         subroutine to strip CTRLs
47          bra.s    main2         look for more files
48
49  * finished all input files, terminate task
50
51  done    move.1   #0,d0         show normal termination
52          sys      term
53
```

```
54
55
56  ************************************************
57
58
59  * subroutine to strip meaningless control characters
60  * from the file specified by file descriptor in "ifd*
61  * and write result to standard output.
62
63
64
65  *begin by reading a buffer full
66
67  strip   move.l   ifd,d0       get input file descriptor
68          sys      read, buffer,BUFSIZ    read buffer full
69          bes.s    rderr        branch if read error
70
71  * check for end of file (0 characters read)
72
73          tst.l    d0           end of input file?
74          beq.s    strip9       exit if so
75
76  * do actual stripping of control characters.  This will
77  * be done in place in the buffer by collapsing the data
78  * as meaningless control characters are stripped.
79
80          move.l   #buffer,a0   point to source buffer
81          move.l   a0,a1        point a1 to destination buffer
82          bra.s    strip6       enter DBcc loop
83  strip4  move.b   (a0)+,d1     get a character into d1
84          cmp.b    $#1F,d1      a control character?
85          bhi.s    strip5       go keep character if not
86          cmp.b    #$0D,d1      a carriage return
87          beq.s    strip5       keep if so
88          cmp.b    #$09,d1      a tab?
89          bne.s    strip6       if not, don't keep
90  strip5  move.b   d1,(a1)+     put char. in buffer
91  strip6  dbra     d0,strip4    decrement count; loop if more
92
93  * finished stripping, a1 points to end of buffer of
94  * stripped data ready to be written
95
96          sub.l    #buffer,a1   find no. of chars to write
97          move.l   a1,wrtcnt    store in parameters
98          move.l   #1,d0        write to standard output
99          sys      ind,iwrite   do indirect write
100         bes.s    writerr      branch if error
101         bra.s    strip        go read another section
102
103 strip9 rts                    exit routine
104
105
106
```

```
107
108
109 ******************************************
110
111 * error handling routines
112
113 opnerr  move.l   d0,-(a7)     save error status on stack
114         move.l   #2,d0        standard error output
115         sys      write,opners,opner1
116         bra.s    err
117 rderr   move.l   d0,-(a7)     save error status on stack
118         move.l   #2,d0        standard error output
119         sys      write,rderrs,rderr1
120         bra.s    err
121 wrterr  move.l   d0,-(a7)     save error status on stack
122         move.l   #2,d0        standard error output
123         sys      write,wrterr,wrter1
124
125 err     move.l   (a7)+,d0     pull error status from stack
126         sys      term         exit program
127
128
129 opners  fcc      "Can't open input file.",$d,0
130 opner1  equ      *-opners
131 rderrs  fcc      'Error reading input file.',$d,0
132 rderr1  equ      *-rderrs
133 wrters  fcc      'Error writing output file.',$d,0
134 wrter1  equ      *-wrters
135
136
137 ****************************************
138
139 * temporary storage and buffers
140
141         data                   begin data segment
142
143 * indirect open system call parameters
144 iopen   dc.w     open          open function code
145 opname  dc.l     0             name of file to open
146 opmode  dc.l     0             open mode 1 (reading)
147
148 * indirect write system call parameters
149 iwrite  dc.w     write         write function code
150 wrtbuf  dc.l     buffer        buffer to write from
151 wrtcnt  dc.l     0             byte count to write
152
153 ifd     ds.l     1             input file descriptor
154
155         ds.b     512-24        reserve up to 512-byte boundary
156 buffer  equ      *             start on 512-byte boundary
157 BUFSIZ  equ      $1000-512     multiple of 512 bytes
158         ds.b     BUFSIZ        reserve space for buffer
159
160
161         end      start
```

Section 5

THE ASSEMBLER AND LINKING LOADER

INTRODUCTION

The 4404's assembler supports conditional assembly as well as numerous other directives for convenient assembler control. The assembler executes in two passes and can accept any size file so long as sufficient memory is installed to contain the symbol table. Output from the assembler is in the form of a relocatable object file.

This section describes the operation and use of the Assembler and Linking Loader. The Assembler accepts most of the Motorola standard mnemonics for instructions, and fully supports the 68000/68010 instruction set. This section describes differences between the Motorola standard for instructions and those supported by the assembler.

This section is not intended to teach the reader assembly language programming nor the full details of the 68000 instruction set. It assumes the user has a working knowledge of assembly language programming and a manual describing the 68000 instruction set and addressing modes in full.

Throughout this section angle brackets ("<" and ">"). are often used to enclose the description of a particular item. The angle brackets to show that it is a single item even though the description may require several words. In addition, square brackets ("[" and "]") are used to enclose an optional item.

Details of the instruction set, assembler syntax, and addressing modes were obtained from "M68000 16/32-Bit Microprocessor Programmer's Reference Manual", Copyright 1984 by Motorola Incorporated.

INVOKING THE ASSEMBLER

Assembler text files must be standard text files with no line numbers or control characters (except for carriage returns and tabs). Once you have both the assembler and the edited source file on a disk or disks which are inserted in a powered-up system, you are ready to begin.

The Command Line

The minimum command line necessary to assemble a source file is:

```
++ asm sourcefile
```

When parameters are omitted, the assembler will assume default parameters. Two types of output are available from the assembler: object code output and assembled source listing output. (The options regarding the assembled source listing output will be described a little later.) Object code is written into a operating system file. It is also possible to disable production of the object code file. Since no specifications are made concerning object code output in the above example, the assembler will assume the default case, which is to produce an object file. Since no name was specified, the object file will assume the same name as the input source file specified but with the characters ".r" appended. If there is not room to append those two characters, the last one or two characters of the input file name will be truncated to make room. In our above example, the created binary file would be named "sourcefile.r". Should a file exist with the same name, it will be automatically deleted with no prompting.

If you wish to create an object file with another name, you may do so by placing the desired file specification on the command line as follows:

```
++ asm sourcefile +o=objectfile
```

The "+o=" is an option to the assembler which specifies that an object file is being created with the specified name. This example would produce an object file named "objectfile". Again, if a file by that name already existed, it would be deleted to permit creation of the new object file.

Multiple Input Source Files

The 4404 assembler is capable of accepting more than one file as the source for assembly. If multiple input files are specified, they are read in the calling order and assembled together to produce a single output file. This permits the user to break source programs down into more convenient size source files which may then be assembled into one object file. As mentioned, the files are read sequentially in the calling order with the last line of source from the current file being followed immediately by the first line of the ensuing file. All "end" statements in the source are effectively ignored and the assembly is terminated when the last line of the last source file is read.

There are two ways to specify multiple input files to the assembler: by entering the name of each file and by a match list in a file specification. Entering each filename would look like this:

```
++ asm file1 file2 file3 file4
```

Using a match list in the file specification we might have:

```
++ asm file[1-4]
```

In this example, the square brackets do not denote an optional item, but rather are the method of specifying a list of match characters. Both of the above examples would produce the same result. Note that in these examples an object file would be created by default and would be called "file1.r" (the name is taken from the first input file). As before, we can also specify an object file name as follows:

```
++ asm file1 file2 file3 file4 +o=command
```

which would result in an object file called "command".

Specifying Assembly Options

Now we shall go one step further and add a set of single character option flags which may be set on the command line as follows:

```
++ asm sourcefile +options
```

The plus sign is required to separate the option(s) from the file specification(s). In this example, the word "options" following the plus sign represents a single character option flag or list of character option flags which either enable or disable a particular option or options. In all cases, they reverse the sense of the particular option from its default sense. Any number of options may be specified and they may be specified in any order. There may not be spaces within the option list.

Following is a list and description of the available options:

+b Do not create a binary file on the disk, even if an binary file name is specified. This is useful when assembling a program to check for errors before the final program is completed or when obtaining a printed source listing.

+e Suppress end summary information. At the end of the assembly, the assembler may report the size of the segments and the total count of errors, warnings and excessive jumps. Often the user does not wish to have any output generated at all; the +e option will suppress this summary information. If this is used without selecting the +l and +s options, then it is

possible that no listing output will be generated. However, if there are any errors reported in the module, this summary information will not be suppressed.

+f Disables the auto-fielding feature of the assembler such that assembled output lines appear in the exact form as found in the input file.

+F Enable debug or "fix" mode. There are two forms of line comments. One begins with an asterisk (*) the other with a semicolon (;), both in the first column of the source line. If the comment begins with a semicolon, the +F option will instruct the assembler to ignore the semicolon and process the line as though the semicolon never existed. The asterisk in the first column of a source line will always denote a comment regardless of the state of this option.

+l Produce the assembled listing output. If specified, the assembler will output each line as it is assembled in the second pass, honoring the 'lis' and 'nol' options (see the 'opt' directive). Those lines containing errors will always be printed, regardless of whether or not this option is specified.

+L Produce a listing of the file during the first pass of the assembler. The assembler prints unformatted lines (exactly as read) to standard output.

+n Enables the printing of decimal line numbers on each output line. These numbers are the consecutive number of the line as read by the assembler. Error lines are always output with the line number, regardless of the state of this option.

+s Produce the symbol table output. If this option is specified, the assembler will produce a sorted symbol table at the end of an assembly. Note that the 'l' option will not produce the symbol table output, just the source listing. In the symbol table, global symbols are preceded by an '*,' and other symbols by a blank.

+S Limit each symbol to only eight characters internally. Normally, the user can define and use symbols that contain 63 unique characters. However, in some cases, it may be necessary to limit the uniqueness of the symbols to only eight characters.

+t Produce object code for the 68000 rather than the 68010. This option affects only the code generation for the "Move from CCR" instruction. Normally the assembler produces the 68010 version of this instruction. If this option is specified, the assembler produces the 68000 "Move from SR" instruction (Privileged on the 68010), in its place.

+u Set all undefined symbols as external. In some cases the user may wish to assemble a module that has some undefined external symbols. The +u option will treat all undefined references as external references. The +u option should not substitute for the good programming practice of listing all external symbols in the operand field of the "extern" directive.

+o file Allows specification of an output object file name (in this example "file").

Order for Specifying Filenames, Options, and Parameters

Input filenames, options, and command line parameters can be specified to the assembler in any order. The assembler scans the input command line twice, once to pick out all options and parameters (they all begin with a plus sign) and then again to pick out all file specifications. Place order is significant only when multiple input files are specified. They will be assembled in the order entered on the calling line.

Sending Output to a Hardcopy Device

The assembler uses the facilities of the 4404's operating system to send the assembled listing to a hardcopy device. The most common means are to route the standard output to a file that may later be spooled or to pipe the standard output to a spooler.

EXAMPLES:

```
++ asm test
```

Assembles a file called "test" and creates an binary file called "test.r" in the same directory. No listing is output (except for any lines with errors) and no symbol table is output.

```
++ asm test +ls
```

Same as before except that assembled listing is output to the terminal, as is the symbol table.

```
++ asm test +o=/bin/test +ls
```

Assembles a file called "test" in the current directory and produces an object file in the "bin" directory called "test". The listing and symbol table are output to the terminal, and if a file by the name of "test" already resides in the "bin" directory, it will be automatically deleted before the assembly starts.

```
++ asm /john/main +bnl
```

This command assembles the file "main" in John's directory but does not produce a binary file. The assembled listing is output with line numbers. No symbol table is printed.

```
++ asm file[1-4] +bln
```

This command assembles all files beginning with "file" and ending with a 1, 2, 3, or 4. No binary or symbol table is output, and line numbers are turned on.

```
++ asm +u dumper +nel
```

This command demonstrates the fact that the filenames, and options can come in any desired order on the command line. The file to be assembled is called "dumper". The assembled listing is output with line numbers. All undefined references will be made external, no summary information will be output, and no symbol table is produced.

ASSEMBLER OPERATION & SOURCE LINE COMPONENTS

The 4404 assembler is a two-pass assembler. In Pass One a symbolic reference table is constructed and, in Pass Two the code is actually assembled, and a listing and object code are produced if requested. The source may be supplied in free format, as described below. Each source line consists of the actual source statement, terminated with a carriage return (OD hex). The source must be comprised of ASCII characters with their parity or 8th bit cleared to zero. Special meaning is attached to many of these characters as will be described later. Control characters ($00 to $FF) other than the carriage return ($OD) and horizontal tab ($09) should not be in the actual source statement part of the line. Their inclusion in the source statement will produce undefined results.

Each source line consists of up to four fields: Label, Opcode, Operand, and Comment. With two exceptions, every line must have an opcode while the other fields may or may not be optional. These two exceptions are:

1. "Comment Lines" may be inserted anywhere in the source and are ignored by the assembler during object code production. Comment lines may be either of two types:

    a. Any line beginning with an asterisk (hex 2A) or semicolon (hex 3B) in column one.

    b. A null line or a line containing only a carriage return. While this line can contain no text, it is still considered a comment line as it causes a space in the output listing.

2. Lines which contain a label but no opcode or operand field.

SOURCE STATEMENT FIELDS

The following pages describe the four source statement fields and their format specifications. The fields are free format which means there may be any number of spaces separating each field. In general, no spaces are allowed within a field.

Label or Symbol Field

This field may contain a symbolic label or name that is assigned the instruction's address and may be called upon throughout the source program.

1. Ordinary Labels

    a. The label must begin in column 1 and must be unique. Labels are optional. If the label is to be omitted, the first character of the line must be a space.

    b. A label may consist of letters (A-Z or a-z), numbers (0-9), or an underscore (_ or 5F hex). Note that upper and lower case letters are not considered equivalent. Thus "ABC" is a different label from "Abc".

    c. Every label must begin with a letter or underscore.

    d. Labels may be of any length, but only the first 63 characters are significant.

    e. The label field must be terminated by a space, tab, or a return.

2. Local Labels

a. Local labels follow many of the same rules as ordinary labels. They must begin in column one and they must be terminated by a space, tab or return.

b. Local labels consist of a number from 0 to 99. These numbers may be repeated as often as desired in the same source module; they need not be in numerical order. Note that the labels "00" and "0", "01" and "1", etc., are unique labels.

c. Local labels may be treated as ordinary labels; however, they cannot be global or external. They may not be used in the label field of an "equ" or "set" directive.

d. Local labels are referenced by using the local label number terminated with an 'f' for first forward reference found or a for the first backward reference found. A backward or forward reference can never refer to the same line that it is found on. For example,

```
2   beq 2f   "2f" => next occurrence of "2"
2   jsr xx   both branches point here
2   bra 2b   "2b" => previous occurrence of "2"
```

e. Local labels should be used primarily (but not necessarily exclusively) for branching or jumping around some sections of code. In most cases, branching around a few lines of code does not warrant the use of an ordinary label. When making a reference to a nearby location in the program there is often no appropriate name with much significance; therefore, programmers have tended to use symbols like l1,l2, etc. This can lead to the danger of using the same label twice. Local labels have freed the programmer from the necessity of thinking of a symbolic name of a location. Furthermore, local labels require less storage internally and lookup is faster than with ordinary labels. A maximum of 500 local labels may be used in one module.

Opcode Field

This field contains the opcode (mnemonic) or a pseudo-op. It specifies the operation that is to be performed. The pseudo-ops recognized by this assembler are described later in this section.

1. The opcode is made up of letters (A-Z or a-z). In this field, upper and lower case may be used interchangeably.

2. This field must be terminated by a space or tab if there is an operand or by a space, tab, or return if there is no operand.

3. The opcode may have a length specification associated with it. This length specification indicates whether the operation is to take place on bytes, words, or long words. The default is words. The specification consists of a period followed by one of the letters "b", "w", "l", or "s". Upper case letters are also permitted. The following summarizes the specifications:

   b or .B bytes (8-bits)
   w or .W words (16-bits, the default)
   l or .L long words (32-bits)
   s or .S short specification (for branches)

Operand Field

The operand provides data or address information required by the opcode. This field may or may not be required, depending on the opcode. Operands are generally combinations of register specifications and mathematical expressions. See the heading of Expressions, later in this section for the rules for forming valid expressions.

1. The operand field can contain no spaces or tabs.

2. This field is terminated with a space, tab, or return.

3. Any of several types of data may make up the operand: register specifications, numeric constants, symbols, ASCII literals.

Comment Field

The comment field may be used to insert comments on each line of source. Comments are for the programmer's convenience only and are ignored by the assembler.

1. The comment field is always optional.

2. This field must be preceded by a space or tab.

3. Comments may contain any characters from SPACE (hex 20) through DELETE (hex 7F) and the tab character.

3. This field is terminated by a carriage return.

REGISTER SPECIFICATION

Many opcodes require that the operand following them specify one or more registers. Both lower and upper case are allowed. The following are possible register names:

D0-D7	Data Registers
A0-A7	Address Registers
A7, SP	System stack pointer of the active system state
USP	User stack pointer
CCR	Condition Code Register (Part of SR)
SR	Status Register
VBR	Vector Base Register (68010)
SFC	Source Function Code Register (68010)
DFC	Destination Function Code Register (68010)

EXPRESSIONS

Many operands must include an expression. This expression may be one or more items combined by any of four operator types: arithmetic, logical, relational, and shift.

Expressions are always evaluated as full 32-bit operations. If the result of the operation is to be fewer bits, the assembler truncates the upper part.

An expression must not contain any embedded spaces or tabs.

ITEM TYPES

The item or items in an expression may be any of the four types listed below. These may stand alone or may be intermixed by the use of the operators.

1. NUMERICAL CONSTANTS: Numbers may be supplied to the assembler in any of the four number bases shown below. The number given will be converted to 32 bits truncating any numbers greater than that. If smaller numbers are required, the 32-bit number will then be further truncated to the proper size. To specify which number base is desired, the programmer must supply a prefix character to a number.

BASE	PREFIX	CHARACTERS ALLOWED
Decimal	none	0 thru 9
Binary	%	0 or 1
Octal @	0 thru 7	
Hexadecimal	$	0 thru 9, A thru F

   If no prefix is assigned, the assembler assumes the number to be decimal.

2. ASCII CONSTANTS: ASCII constants may be specified in expressions by enclosing the string in single or double quotation marks. The string must consist of one to four characters, depending on the desired size attribute. The specified characters may not include control characters (must be between 20 hex and 7F hex inclusive).

3. LABELS: An expression may contain labels which have been assigned some address, constant, relocatable or external value. As described above under the label field, a label consists of letters, digits, and underscores beginning with a letter or underscore. The label may be of any length, but only the first 63 characters are significant. Any label used in the operand field must be defined elsewhere in the program. Local labels may also be used in the operand field. None of the standard register specifications should be used as a label.

4. PC DESIGNATOR: The asterisk (*) has been set aside as a special PC designator (Program Counter). It may be used in an expression just as any other value and is equal to the address of the current instruction. The value of the PC designator is relocatable in the text, data or bss segments; its value is given at load time.

Types of Expressions

Three types of expressions are possible in the 4404 assembler: absolute, relocatable and external expressions.

Absolute Expressions. An expression is absolute if its value is unaffected by program relocation. An expression can be absolute, even though it contains relocatable symbols, under both of the following conditions:

1. The expression contains an even number of relocatable elements.

2. The relocatable elements must cancel each other. That is, each relocatable element (or multiple) in a segment must be canceled by another element (or multiple) in the same segment. In other words, pairs of elements in the same segment must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

For example, text1 and text2 are two relocatable symbols in the text segment; the following examples are absolute expressions.

```
text1-text2
5*(text1-text2)
```

Relocatable Expressions. An expression is relocatable if its value is affected by program relocation in a relocatable module. A relocatable expression consists of a single relocatable symbol or, under all three of the following conditions, a combination of relocatable and absolute elements.

1. The expression does not contain an even number of relocatable elements.

2. All the relocatable elements but one must be organized in pairs that cancel each other. That is, for all but one segment, each relocatable element (or multiple) in a segment must be canceled by another element (or multiple) in the same block.

3. The uncancelled element can have either positive or negative relocation.

For example, text1 and text2 are symbols from the text segment, data1 and data2 are symbols from the data segment, and bss1 and bss2 are symbols from the bss segment; the following examples are relocatable:

```
-bss2+3*5+(data2-data2)  negative relocation from bss segment
text1+(data1-data2)+(bss2-bss1)  relocation from text segment
data1-(bss2-bss1)                relocation from data segment
*          (PC Designator)   relocation from current segment
```

External Expressions. An expression is external if its value depends upon the value of a symbol defined outside of the current source module. An external expression can consist of a single external symbol, or, under both of the following conditions, an external expression may consist of an external symbol, relocatable elements and absolute elements:

1. The expression contains an even number of relocatable elements

2. The relocatable elements must cancel each other. That is, each relocatable element (or multiple) in a segment must be canceled by another element in the same segment. In other words, pairs of elements in the same segment must have signs that oppose each other.

For example, if ext1 is an external symbol, text1, text2, data1, data2, bss1, bss2 all have the same meaning as above in the previous examples; then the following examples are external:

```
(text1-text2)+ext1-(data2-data1)
5+ext1-3
3/(text2-text1)-ext1
```

Expression Operators

Operators permit operations such as addition or division to take place during the assembly, and the result becomes a permanent part of your program. Many of these operators will only apply to absolute symbols and expressions. It does not make sense to multiply a relocatable or external value at assembly-time! Only the + and - operators can apply to relocatable and external symbols and expressions.

Arithmetic Operators. The arithmetic operators are:

Operator	Meaning
+	Unary or binary addition
-	Unary or binary subtraction
*	Multiplication
/	Division (any remainder is discarded)

Logical Operators. The logical operators are:

Operator	Meaning
&	Logical AND operator
¦	Logical OR operator
!	Logical NOT operator
>>	Shift right operator
<<	Shift left operator

The logical operations are full 32-bit operations. In other words for the AND operation, every bit from the first operand or item is individually ANDed with its corresponding bit from the second operand or item. The shift operators shift the left term the number of places indicated by the right term. Zeroes are shifted in and any bits shifted out are lost.

Relational Operators. The relational operators are:

Operator	Meaning
=	Equal
<	Less than
>	Greater than
<>	Not equal
<=	Less than or equal
>=	Greater than or equal

The relational operations yield a true-false result. If the evaluation of the relation is true, the resulting value be all ones. If false, the resulting value will will be all zeros. Relational operations are generally used in conjunction with conditional assembly, as shown in that discussion.

Operator Precedence. Certain operators take precedence over others in an expression. This precedence can be overcome by the use of parentheses. If there is more than one operator of the same precedence level, and no parentheses indicate the order in which they should be evaluated, then the operations are carried out in left to right order.

The following list classifies the operators in order of precedence (highest priority first):

1) Parenthesized expressions
2) Unary + and -
3) Shift operators
4) Multiply and Divide
5)
6) Relational Operators
7) Logical NOT Operator
8) Logical AND and OR Operators

INSTRUCTION SET DIFFERENCES

This discussion describes the differences in the instruction mnemonics accepted by the assembler and the Motorola standard. The standard is assumed to be that defined in the "MC68000 16-Bit Microprocessor User's Guide," published by Motorola Semiconductor Products, Inc. It is assumed that the reader is familiar with the contents of the "Instruction Set Details" portion of that manual. In particular, the user should be familiar with the description of the assembler syntax that accompanies the discussion of the individual instructions.

The assembler recognizes the standard instruction set with the exception of some of the so-called "variations". Having a specific opcode for these variations is not necessary, because the assembler can infer their existence from an analysis of the operands and generate the proper code. This relieves the programmer from the need for remembering the opcodes, and the particulars of each. The variations that are handled in this manner are: address, quick, and immediate. Note that the "extend" variation is still supported. Thus, the following instructions are not specifically recognized by the Assembler:

ADDA, ADDQ, ADDI	Use ADD instead
ANDI	Use AND instead
CMPA, CMPI, CMPM	Use CMP instead
EORI	Use EOR instead
MOVEA, MOVEQ	Use MOVE instead
ORI	Use OR instead
SUBA, SUBQ, SUBI	Use SUB instead

Remember that even though these mnemonics are not recognized, the assembler can and does generate code for address, quick, and immediate instructions. The proper instruction is selected automatically after analyzing the operands.

The default data size is "word". Instructions that can manipulate more than one size of data item may be modified by postfixing a data length specification to the opcode. The data length specifications are:

l or .L	For long word (32 bits)
w or .W	For word (16 bits, the default)
b or .B	

THE INSTRUCTION SET

PROGRAMMING MODEL

The 68000 microprocessor has 16 32-bit general purpose registers, a 32-bit program counter, and an 8-bit condition code register. The registers are:

D0-D7	Data registers
A0-A6	Address registers
A7	Stack pointer (Also available as "SP")
CCR	The condition code register
PC	Program counter

The supervisor programmer's model also includes:

SSP	Supervisor stack pointer
SR	Status register
VBR	Vector base register (68010)
SFC	Source function code register (68010)
DFC	Destination function code register (68010)

The data registers can be used for 8-bit, 16-bit, or 32-bit operations. The address registers can be used for 16-bit or 32-bit operations as can base address registers. All registers can be used as index registers.

ADDRESSING MODES

Twelve addressing modes are available on the 68000, divided into six categories. Each assembled instruction takes a minimum of one word of storage. The different addressing modes use different amounts of storage depending on what information is needed to form the specified effective address. The maximum storage required by any addressing mode is two words. The amount of extra storage required by an addressing mode, called extension words, is stated with each description. In the descriptions, registers are specified as "Dn", "An", or "Rn", referring to data register 'n', address register or either data or address register respectively. The 'D' and 'A' in the register specification can be either upper- or lowercase. must be number from to inclusive.

1. Data Register Direct The operand is in the data register specified.

   Assembler Syntax: Dn

   Example: EXT.L DO Sign-extends data register 0 to 32-bits.

2. Address Register Direct The operand is in the address register specified.

   Assembler Syntax: An

   Example: ADD.L A1,A2 Add the contents of address register 1 to address register 2.

3. Address Register Indirect The address of the operand is in the address register specified.

   Assembler Syntax: (An)

   Example: SUB.L D5,(A4) Subtract the contents of data register 5 from the long operand at the address in address register 4.

4. Address Register Indirect With Postincrement

   The address of the operand is in the address register specified. After the operand address is used, it is incremented by one, two, or four depending on whether the size of the operand is byte, word, or long.

   Assembler Syntax: (An)+

Example: CLR.W (A5)+ Zero the word at the address in address register five, then increment the register by two.

5. Address Register Indirect With Predecrement

The address of the operand is in the address register specified. Before the operand address is used, it is decremented by one, two, or four depending on whether the size of the operand is byte, word, or long.

Assembler Syntax: -(An)

Example: CLR.B -(A3) Decrement address register 3 by one, then zero the byte at the address now in address register three.

6. Address Register Indirect With Displacement

The address of the operand is the sum of the address in the address register and the sign-extended displacement specified. The displacement must be an absolute, 16-bit expression. This addressing mode requires one word of extension.

Assembler Syntax: displacement(An)

Example: MOVE.L 6(A0),D1 Move the four bytes at address register zero plus six into data register one.

7. Address Register Indirect With Index

The address of the operand is the sum of the address in the address register, the sign-extended displacement specified, and the contents of the index register. The displacement need not be specified, in which case it is assumed to be zero. The displacement, if specified, must be an absolute, 8-bit expression. Either the entire 32 bits of the index register (".L" extension), or the sign-extended low-order, 16 bits may be used (".W" extension). The default is to use the low-order 16 bits and the 'W' and 'L' may be upper- or lowercase. One word of extension is required by this addressing mode.

Assembler Syntax: displacement(An,Rn.W)
displacement(An,Rn.L)

Example: CLR.L $A(A1,D1.W) Zero the four bytes at address register 1 plus the low-order 16 bits of data register 1 plus $A (10 decimal).

TST.L (A2,A3.L) Test the four bytes at address register 2 plus address register 3.

8. Absolute Short Address

The address of the operand is the absolute or relocatable displacement specified. The 16-bit address is sign-extended before it is used. This addressing mode requires one word of extension. The assembler requires that only program labels be used with the ":W" extension.

Assembler Syntax: label:W

Example: JSR sqrt:W Jump to the subroutine "sqrt" using a 16-bit address.

9. Absolute Long Address

The address of the operand is the absolute or relocatable displacement specified. This addressing mode requires two words of extension.

Assembler Syntax: label or displacement

Example: JSR sqrt Jump to the subroutine "sqrt" using a 32-bit address.

JSR $400300 Jump to the subroutine at hex location 400300.

10. Program Counter With Displacement

The address of the operand is the sum of the address in the program counter and the sign-extended displacement integer. The displacement must be a 16-bit expression and is formed by subtracting the value of the program counter from the address of the label specified. The label specified must be relocatable, and must be in the same segment as the current program counter. The operand field for branch instructions requires only a label; all other instructions wishing to use this addressing mode must follow the syntax below, to distinguish this addressing mode from the absolute long addressing mode. This addressing mode requires one word of extension.

Assembler Syntax: label(PC)

Example: MOVE.L table(PC),A1 Move the four bytes at the program counter plus the difference between the address of "table" and the program counter into address register 1.

11. Program Counter With Index

The address of the operand is the sum of the address in the program counter, the sign-extended displacement integer, and the contents of the index register. The displacement is calculated by subtracting the program counter from the address of the label specified in the instruction. This displacement must be an 8-bit expression. The label specified must be relocatable, and must be in the same segment as the current program counter. Either the entire 32 bits of the index register (".L" extension), or the sign-extended, low-order, 16 bits may be used (".W" extension). The default is to use the low-order 16 bits. The 'W' and 'L' may be upper- or lowercase. One word of extension is required by this addressing mode.

Assembler Syntax: label(PC,Rn.W)
label(PC,Rn.L)

Example: MOVE.L table(PC,D1.L),A3 Move the four bytes at the program counter plus the difference between the address of "table" and the program counter, plus the contents of data register 1 into address register 3.

12. Immediate Data

The operand is the immediate value specified. This addressing mode requires one or two words of extension, depending on the size of the operation.

Assembler Syntax: expression

Example: MOVE.W 4096,D2 Move 4096 into data register 2.

The following table shows how each addressing mode falls into the six categories.

Category	Addressing Modes 1	2	3	4	5	6	7	8	9	10	11	12
Data Addressing	X		X	X	X	X	X	X	X	X	X	X
Control Addressing			X			X	X	X	X	X	X	
Alterable	X	X	X	X	X	X	X	X	X			
Data Alterable	X		X	X	X	X	X	X	X			
Memory Alterable			X	X	X	X	X	X	X			
Control Alterable			X									

THE ASSEMBLER INSTRUCTION SET

Syntax

This sections contains a brief alphabetical listing of all the mnemonics accepted by the assembler. The following notational conventions will be used:

An	Address register 'n'
Dn	Data register 'n'
Rn	Either data or address register 'n'
Rc	Control register, address or data
<disp>	8-, 16- or 32-bit displacement value
<disp(8)>	8-bit displacement value
<disp(16)>	16-bit displacement value
<disp(32)>	32-bit displacement value
<ea>	Effective address
<data>	8-, 16- or 32-bit data value
<data(8)>	8-bit data value
<data(16)>	16-bit data value
<data(32)>	32-bit data value
<vector>	Vector number from 0 through 15
<quick>	A data value from 1 through 8 (quick value)
<label>	A label in the source file

<bit_mask> A 16-bit mask specifying which registers to move in a "MOVEM" instruction. Using the pre-decrement addressing mode, the bit correspondence is:

```
Bit 0  -  Address register 7
Bit 1  -  Address register 6
...
Bit 7  -  Address register 0
Bit 8  -  Data register 7
Bit 9  -  Data register 6
...
Bit 15 -  Data register 0
```

Using all other addressing modes, the bit correspondence is:

```
Bit 0  -  Data register 0
Bit 1  -  Data register 1
...
Bit 7  -  Data register 7
Bit 8  -  Address register 0
...
Bit 15 -  Address register 7
```

Bits are numbered with the rightmost bit being number 0 and the leftmost being number 15.

<register_list>

A register list is used for the "MOVEM" instruction. Register lists can be formed two ways. Registers can be separated by a '/' such as:

```
D1/D3/D5/A2/A3
```

to specify individual registers to be moved. Register lists can also be specified by separating two registers with a '-' such as:

```
D1-D5/A1-A3
```

to specify that registers D1 through D5 inclusive and registers A1 through A3 inclusive should be moved.

Instructions

ABCD Add decimal with extend

```
Assembler Syntax:  ABCD Dy,Dx
                   ABCD -(Ay),-(Ax)
```

ADD Add binary

```
Assembler Syntax:  ADD <ea>,Dn
                   ADD Dn,<ea>
                   ADD <ea>,An
                   ADD  <data>,<ea>
```

Source Effective Address: All addressing modes
Destination Effective Address: Data alterable addressing modes

ADDX Add extended

```
Assembler Syntax:  ADDX Dy,Dx
                   ADDX -(Ay),-(Ax)
```

AND AND logical

```
Assembler Syntax:  AND <ea>,Dn
                   AND Dn,<ea>
                   AND  <data>,<ea>
                   AND  <data(8)>,CCR
                   AND  <data(16)>,SR
```

Source Effective Address: Data addressing modes
Destination Effective Address: Data alterable addressing modes

ASL Arithmetic shift left

```
Assembler Syntax:  ASL Dx,Dy
                   ASL  <quick>,Dn
                   ASL <ea>
```

Source Effective Address: Memory alterable (word only)

ASR Arithmetic shift right

```
Assembler Syntax:  ASR Dx,Dy
                   ASR  <quick>,Dn
                   ASR <ea>
```

Source Effective Address: Memory alterable (word only)

Bcc Branch conditionally
Assembler Syntax: Bcc <label>
Legal Branches:

BCC Branch on carry clear
BCS Branch on carry set
BEQ Branch on equal
BGE Branch on greater or equal
BGT Branch on greater
BHI Branch on high
BHS Branch on high or same (BCC)
BLE Branch on less or equal
BLO Branch on low (BCS)
BLS Branch on low or same
BLT Branch on less than
BMI Branch on minus
BNE Branch on not equal
BPL Branch on plus
BRA Branch always (unconditionally)
BVC Branch on overflow clear
BVS Branch on overflow set
BCHG Test a bit and change

Assembler Syntax: BCHG Dn,<ea>
BCHG <data(8)>,<ea>
Destination Effective Address: Data alterable addressing modes

BCLR Test a bit and clear
Assembler Syntax: BCLR Dn,<ea>
BCLR <data(8)>,<ea>
Destination Effective Address: Data alterable addressing modes

BSET Test a bit and set
Assembler Syntax: BSET Dn,<ea>
BSET <data(8)>,<ea>
Destination Effective Address: Data alterable addressing modes

BSR Branch to subroutine
Assembler Syntax: BSR <label>

BTST Test a bit
Assembler Syntax: BTST Dn,<ea>
BTST <data(8)>,<ea>
Destination Effective Address: Data addressing modes

CHK Check register against bounds
Assembler Syntax: CHK <ea>,Dn
Source Effective Address: Data addressing modes

CLR Clear an operand
Assembler Syntax: CLR <ea>
Source Effective Address: Data alterable addressing modes

CMP Compare
Assembler Syntax: CMP <ea>,Dn
CMP <ea>,An
CMP <data>,<ea>
CMP (Ay)+,(Ax)+
Source Effective Address: All addressing modes
Destination Effective Address: Data alterable addressing modes

DBcc Test condition, decrement, and branch
Assembler Syntax: DBcc Dn,<label>
Legal Decrement and branches:
DBCC Decrement and branch on carry clear
DBCS Decrement and branch on carry set
DBEQ Decrement and branch on equal
DBF Decrement and branch on false (unconditionally)
DBGE Decrement and branch on greater or equal
DBGT Decrement and branch on greater
DBHI Decrement and branch on high
DBLE Decrement and branch on less or equal
DBLS Decrement and branch on low or same
DBLT Decrement and branch on less than
DBMI Decrement and branch on minus
DBNE Decrement and branch on not equal
DBPL Decrement and branch on plus
DBRA Decrement and branch always (DBF)
DBT Decrement and branch on true
DBVC Decrement and branch on overflow clear
DBVS Decrement and branch on overflow set

DIVS Signed divide
Assembler Syntax: DIVS <ea>,Dn
Source Effective Address: Data addressing modes

DIVU Unsigned divide
Assembler Syntax: DIVU <ea>,Dn
Source Effective Address: Data addressing modes

EOR Exclusive OR logical
Assembler Syntax: EOR Dn,<ea>
EOR <data>,<ea>
EOR <data(8)>,CCR
EOR <data(16)>,SR
Destination Effective Address: Data alterable addressing modes

EXG Exchange registers
Assembler Syntax: EXG Rx,Ry

```
EXT       Sign extend
          Assembler Syntax:  EXT Dn

ILLEGAL   Illegal instruction
          Assembler Syntax:  ILLEGAL

JMP       Jump
          Assembler Syntax:  JMP <ea>
          Source Effective Address:      Control addressing modes

JSR       Jump to subroutine
          Assembler Syntax:  JSR <ea>
          Source Effective Address:      Control addressing modes

LEA       Load effective address
          Assembler Syntax:  LEA <ea>,An
          Source Effective Address:      Control addressing modes

LINK      Link and allocate
          Assembler Syntax:  LINK An, <disp(16)>

LSL       Logical shift left
          Assembler Syntax:  LSL Dx,Dy
                             LSL  <quick>,Dn
                             LSL <ea>
          Source Effective Address:      Memory alterable (word only)

LSR       Logical shift right
          Assembler Syntax:  LSR Dx,Dy
                             LSR  <quick>,Dn
                             LSR <ea>
          Source Effective Address:      Memory alterable (word only)

MOVE      Move data from source to destination
          Assembler Syntax:  MOVE <ea>,<ea>
                             MOVE CCR,<ea>
                             MOVE <ea>,CCR
                             MOVE <ea>,SR
                             MOVE SR,<ea>
                             MOVE USP,An
                             MOVE An,USP
          Source Effective Address:      All addressing modes
          Destination Effective Address: Data alterable addressing modes

          On "MOVE TO CCR/SR":
          Source Effective Address:      Data addressing

MOVEC     Move to/from control register
          Assembler Syntax:  MOVEC Rc,Rn
                             MOVEC Rn,Rc
```

MOVEM Move multiple registers
Assembler Syntax: MOVEM <register_list>,<ea>
MOVEM <ea>,<register_list>
MOVEM <bit_mask>,<ea>
MOVEM <ea>,<bit_mask>
Source Effective Address: Control addressing and postincreme
Destination Effective Address: Control alterable and predecrement

MOVEP Move peripheral data
Assembler Syntax: MOVEP Dx,<disp(16)>(Ay)
MOVEP <disp(16)>(Ax),Dy

MOVES Move to/from address space
Assembler Syntax: MOVES Rn,<ea>
MOVES <ea>,Rn
Source Effective Address: Memory alterable addressing modes
Destination Effective Address: Memory alterable addressing modes

MULS Signed multiply
Assembler Syntax: MULS <ea>,Dn
Source Effective Address: Data addressing modes

MULU Unsigned multiply
Assembler Syntax: MULU <ea>,Dn
Source Effective Address: Data addressing modes

NBCD Negate decimal with extend
Assembler Syntax: NBCD <ea>
Source Effective Address: Data alterable addressing modes

NEG Negate
Assembler Syntax: NEG <ea>
Source Effective Address: Data alterable addressing modes

NEGX Negate with extend
Assembler Syntax: NEGX <ea>
Source Effective Address: Data alterable addressing modes

NOP No operation
Assembler Syntax: NOP

NOT Logical complement
Assembler Syntax: NOT <ea>
Source Effective Address: Data alterable addressing modes

OR — Inclusive OR logical

```
Assembler Syntax:  OR <ea>,Dn
                   OR Dn,<ea>
                   OR  <data>,<ea>
                   OR  <data(8)>,CCR
                   OR  <data(16)>,SR
Source Effective Address:       Data addressing modes
Destination Effective Address: Data alterable addressing modes
```

PEA — Push effective address

```
Assembler Syntax:  PEA <ea>
Source Effective Address:       Control addressing modes
```

RESET — Reset external devices

```
Assembler Syntax:  RESET
```

ROL — Rotate left

```
Assembler Syntax:  ROL Dx,Dy
                   ROL  <quick>,Dn
                   ROL <ea>
Source Effective Address:       Memory alterable (word only)
```

ROR — Rotate right

```
Assembler Syntax:  ROR Dx,Dy
                   ROR  <quick>,Dn
                   ROR <ea>
Source Effective Address:       Memory alterable (word only)
```

ROXL — Rotate left with extend

```
Assembler Syntax:  ROXL Dx,Dy
                   ROXL  <quick>,Dn
                   ROXL <ea>
Source Effective Address:       Memory alterable (word only)
```

ROXR — Rotate right with extend

```
Assembler Syntax:  ROXR Dx,Dy
                   ROXR  <quick>,Dn
                   ROXR <ea>
Source Effective Address:       Memory alterable (word only)
```

RTD — Return and deallocate parameters

```
Assembler Syntax:  RTD  <disp(16)>
```

RTE — Return from exception

```
Assembler Syntax:  RTE
```

RTR — Return and restore condition codes

```
Assembler Syntax:  RTR
```

RTS — Return from subroutine
Assembler Syntax: RTS

SBCD — Subtract decimal with extend
Assembler Syntax: SBCD Dy,Dx
SBCD -(Ay),-(Ax)

Scc — Set according to condition
Assembler Syntax: Scc <ea>

Legal Sets:
SCC Set on carry clear
SCS Set on carry set
SEQ Set on equal
SF Set on false
SGE Set on greater or equal
SGT Set on greater
SHI Set on high
SLE Set on less or equal
SLS Set on low or same
SLT Set on less than
SMI Set on minus
SNE Set on not equal
SPL Set on plus
ST Set on true (unconditionally)
SVC Set on overflow clear
SVS Set on overflow set

Source Effective Address: Data alterable addressing modes

STOP — Load status register and stop
Assembler Syntax: STOP <data(16)>

SUB — Subtract binary
Assembler Syntax: SUB <ea>,Dn
SUB Dn,<ea>
SUB <ea>,An
SUB <data>,<ea>
Source Effective Address: All addressing modes
Destination Effective Address: Data alterable addressing modes

SUBX — Subtract with extend
Assembler Syntax: SUBX Dy,Dx
SUBX -(Ay),-(Ax)

SWAP — Swap register halves
Assembler Syntax: SWAP Dn

TAS Test and set an operand
Assembler Syntax: TAS <ea>
Source Effective Address: Data alterable addressing modes

TRAP Trap
Assembler Syntax: TRAP <vector>

TRAPV Trap on overflow
Assembler Syntax: TRAPV

TST Test an operand
Assembler Syntax: TST <ea>
Source Effective Address: Data alterable addressing modes

UNLK Unlink
Assembler Syntax: UNLK An

Convenience Mnemonics

CLC	Clear carry condition code bit
CLN	Clear negative condition code bit
CLV	Clear overflow condition code bit
CLX	Clear extend condition code bit
CLZ	Clear zero condition code bit
SEC	Set carry condition code bit
SEN	Set negative condition code bit
SEV	Set overflow condition cod
SEX	Set extend condition code bit
SEZ	Set zero condition code bit

STANDARD DIRECTIVES OR PSEUDO-OPS

Besides the standard machine language mnemonics, the assembler supports several directives or pseudo-ops. These are instructions for the assembler to perform certain operations, and are not directly assembled into code. There are three types of directives in this assembler: those associated with conditional assembly, those associated with macros, and those which generally can be used anywhere which we shall call "standard directives".

The standard directives are:

dc	log
ds	opt
equ	pag
err	rab
even	rmb
fcb	rzb
fcc	set
fdb	spc
fqb	sttl
info	sys
lib	ttl

Other types of directives are explained in other sections, but are listed here for completeness:

Conditional Directives	Relocation Directives	
if	base	end
ifn	bss	extern
else	common	global
endif	endcom	name
	data	struct
	define	text
	enddef	

DC

The "dc" or Define Constant directive defines one or more constants in memory. A size specification may be postfixed to the directive to indicate that the constant is to be stored in bytes, words, or long words. The default is "words". If multiple operands are specified, the effect is as though the operands appeared in consecutive "dc" directives. The operands may be actual values (constants or ASCII strings) or expressions. ASCII strings must be enclosed in single quotation marks.

The constant is aligned on the proper boundary, depending on the size specification (byte boundary for ".b", word boundary for ".w", and long word boundary for ".l"). When ASCII strings are specified with a word or long word size specification, the string will be padded on the right with zero bytes if there are not enough characters to exactly fill the last word or long word. If an ASCII string is specified with a byte size specification, and the instruction or directive following the "dc.b" directive requires word or long word alignment, then zeroes will be appended to the character string to force such alignment. Some examples:

```
label1 dc.b 3,7,'String'

label2 dc.w 123,'abc',98      The 'abc' will be padded with
                              a zero byte

dc.l 'a',131072     The 'a' will be padded with 3 zero bytes
```

DS

The "ds" or Define Storage directive reserves areas of memory. The reserved memory is not guaranteed to be initialized in any way. A size specification may be postfixed to the directive to indicate that bytes, words, or long words are to be reserved. If words or long words are specified, the reserved memory will be properly aligned. A single operand indicates how many bytes, words, or long words are to be reserved. If a label is present, its value will be the address of the lowest memory location reserved. If the value of the operand is zero, no space will be reserved; however, alignment will take place if "ds.w" or "ds.l" is specified. Some examples:

```
ds.b 20       reserve 20 bytes
ds   10       reserve 10 words
ds.l 5        reserve 5 long words
ds.l 0        force alignment on long word boundary
```

EQU

The "equ" or Equate directive equates a symbol to the expression given in the operand. No code is generated by this statement. Once a symbol has been equated to some value, it may not be changed at a later time in the assembly. The form of an equate statement is

```
<label>  equ  <nonexternal expression>
```

The label is strictly required in equate statements. Absolute or relocatable expressions are allowed; external expressions are illegal. If the expression is relocatable, both the value and the attribute will be assigned to the label.

ERR

The "err" directive may be used to insert user-defined error messages in the output listing. The error count is also incremented by one. The format is:

```
err  <message to be printed>
```

All text past the "err" directive (excluding leading spaces) is printed as an error message (preceded by three asterisks) in the output listing. Note that the "err" directive line itself is not printed. A common use for the "err" directive is in conjunction with conditional assembly, to report user-defined illegal conditions.

EVEN

The "even" directive is used to force the program counter to an even address (word boundary).

FCB

The 'fcb' or Form Constant Byte directive is used to set associated memory bytes to some value as determined by the operand. 'fcb' may be used to set any number of bytes, as shown below:

```
[<label>]  fcb  <expr. 1>,<expr. 2>,...,<expr. n>
```

<expr. x> stands for some absolute, relocatable or external expression. Each expression given (separated by commas) is evaluated to 8 bits, and the resulting quantities are stored in successive memory locations. The label is optional.

FCC

The 'fcc' or Form Constant Character directive allows the programmer to specify a string of ASCII characters delimited by some non-alphanumeric character such as a single quote. All the characters in the string will be converted to their respective ASCII values and stored in memory, one byte per character. Some examples:

```
label1  fcc  'This is an fcc string'
label2  fcc  .so is this.
        fcc  /Labels are not required./
```

There is another method of using 'fcc' which is a deviation from the standard Motorola definition of this directive. This method allows you to place certain expressions on the same line as the standard 'fcc' delimited string. The items are separated by commas and are evaluated to 8-bit results. In some respects this is like the 'fcb' directive. The difference is that in the 'fcc' directive, expressions must begin with a letter, number or dollar sign, whereas in the cb' directive any valid expression will work. For example, %10101111 is a valid expression for a 'fcb' but not for a 'fcc' since the percent-sign would look like a delimiter and the assembler would attempt to produce 8 bytes of data from 8 ASCII characters which follow (a 'fcc' string). The dollar sign is an exception to allow hex values such as $0D (carriage return) to be inserted along with strings. Some examples:

```
intro  fcc  'This string has CR & LF',$D,$A
       fcc  'string 1',0,'string 2'
       fcc  $04,extlabel,/delimited string/
```

Note that more than one delimited string may be placed on a line as in the second example.

FDB

The "fdb" or Form Double Byte directive is used to create 16 bit constants in memory. It is exactly like the "fcb" directive except that 16 bit quantities are evaluated and stored in memory for each expression given. The form of the statement is:

label>] fdb <expr. 1>,<expr. 2>,...,<expr. n>

Again, the label field is optional. The generated data is guaranteed to be on a word boundary (see the "dc" directive).

FQB

The "fqb" or Form Quad Byte directive is used to create 32-bit constants in memory. It is exactly like the "fdb" directive, except that 32-bit quantities are evaluated and stored in memory for each expression given. The form of the statement is:

label>] fdb <expr. 1>,<expr. 2>,...,<expr. n>

Again, the label field is optional. The generated data is guaranteed to be on a word boundary (see the "dc" directive).

INFO

The "info" directive allows the user to store textual comments in a binary file. A 4404 user can execute the command 'info' and view the text the screen. The assembler's 'info' directive places all text following the 'info' command (excluding leading spaces) into a temporary file called '/tmp/asmbinfoxxxxx', where xxxxx represents the current task number. At the end of the assembly, all text stored in this temporary file is appropriately copied into the normal binary file, and the temporary file is then deleted. Syntax is as follows:

```
info  This is a comment for the binary file.

info  It is a convenient way of inserting version nos.

info  Version X.XX - Released XX/XX/XX
```

Any number of 'info' directives may be inserted at any point in the source listing. No label is allowed, and no actual binary code is produced.

LIB

The "lib" or Library directive allows the user to specify an external file for inclusion in the assembled source output. Under normal conditions, the assembler reads all input from the file(s) specified on the calling line. The 'lib' directive allows the user to temporarily obtain the source lines from some other file. When all the lines in that external file have been read and assembled, the assembler resumes reading of the original source file. The proper syntax is:

```
lib  <file spec>
```

where <file spec> is a standard 4404 file specification.

The assembler first looks for the specified file in the current directory. If the file isn't found in the current directory, the assembler then looks for a directory named "lib" in the current directory. If it finds such a directory, the assembler attempts to find the specified file in that "lib" directory. If not found there, the assembler makes a third and final attempt to find the specified file by looking in the directory "/lib". If the file is not found in any of these three directories, the assembler gives up and reports an error.

Any "end" statements found in the file called by the 'lib' directive are ignored. The "lib" directive line itself does not appear in the output listing. Any number of "lib" instructions may appear in a source listing. It is also possible to nest 'lib' files up to 4-6 levels.

LOG

The "log" directive is used to calculate the log, base 2, of an absolute expression. The result is 32 bits. The statement acts like a "set" statement, in that the label specified can be redefined with other "log" directives or "set" directives. The form of the statement is:

```
[<label>]  log  <absolute expression>
```

The label field is strictly required.

OPT

The "opt" or Option directive allows the user to choose from several different assembly options. These options are generally related to the format of the output listing and object code. The options which may be set with this command are listed below. The proper form of this instruction is:

```
opt  <option 1>,<option 2>,...,<option n>
```

Note that any number of options may be given on one line if separated by commas. No label is allowed, and no spaces or tabs may be embedded in the option list. The options are set during Pass Two. If contradicting options are specified, the last one on the command line takes precedence. If a particular option is not specified, the default case for that option takes effect. The default cases are signified below by an asterisk.

The allowable options are:

```
con   print conditionally skipped code
noc*  suppress conditional code printing

lis*  print an assembled listing
nol   suppress output of assembled listing
```

The "lis" and "nol" options may be used to selectively turn parts of a program listing on or off as desired. If the "+l" command line option is specified, however, the "lis" and "nol" options are overridden and no listing occurs.

PAG

The "pag" directive causes a page eject in the output listing and prints a header at the top of the new page. Note that the "pag" option must be enabled in order for this directive to take effect. It is possible to assign a new number to the new page by specifying such in the operand field. If no page number is specified, the next consecutive number will be used. No label is allowed and no code is produced. The "pag" operator itself will not appear in the listing unless some sort of error is encountered. The proper form is:

```
pag  [<expression>]
```

The expression is optional. The first page of a listing does not include the header and is considered to be page 0. Thus, all options, title, and subtitle may be set up and followed by a "pag" directive to start the assembled listing at the top of page 1 without the option, title, or subtitle instructions being in the way.

RAB

The 'rab' or Reserve Aligned Bytes directive is used to reserve areas of memory for data storage. The bytes are forced to a word boundary. The number of bytes specified by the expression in the operand are skipped during assembly. No code is produced in those memory location and therefore the contents are undefined at run time. The proper usage is shown here:

[<label>] rab <absolute expression>

The label is optional, and the absolute expression is a 32-bit quantity. "rab" directives found in the text or data segments act like "rzb", and produce code which is guaranteed to be on an even boundary.

RMB

The 'rmb' or Reserve Memory Bytes directive is used to reserve areas of memory for data storage. The number of bytes specified by the expression in the operand are skipped during assembly. No code is produced in those memory locations and therefore the contents are undefined at run time. The proper usage is:

[<label>] rmb <absolute expression>

The label is optional, and the absolute expression is a 32-bit quantity. Any "rmb" directives found in the text or data segments act like "rzb", and produce code.

RZB

The 'rzb' or Reserve Zeroed Bytes directive is used to initialize an area of memory with zeroes. Beginning with the current PC location, the number of bytes specified will be set to zero. The proper syntax is:

[<label>] rzb <absolute expression>

where the absolute expression is a 32-bit expression. This directive does produce object code. Any "rzb" directives found in the bss segment act like "rmb".

SET

The "set" directive sets a symbol to the value of some expression, much as an "equ" directive. The difference is that a symbol may be "set" several times within the source (to different values), but may be "equated" only once. If a symbol is "set" to several values within the source, the current value of the symbol will be the value last "set". The statement form is:

```
<label>  set  <nonexternal expression>
```

The label is strictly required, and no code is generated.

SPC

The "spc" or Space directive inserts the specified number of spaces (line feeds) into the output listing. The general form is:

```
spc  [<space count>[,<keep count>]]
```

The space count can be any number from 0 to 255. If the page option is selected, "spc" will not cause spacing past the top of a new page. The <keep count>, which is optional, is the number of lines to keep together on a page. If there are not enough lines left on the current page, a page eject is performed. If there are <keep count> lines left on the page (after printing <space count> spaces), output will continue on the current page. If the page option is not selected, the <keep count> will be ignored. If no operand is given, the assembler will default to one blank line in the output listing.

STTL

The "sttl" or Subtitle directive is used to specify a subtitle to be printed just below the header at the top of an output listing page. It is specified much as the "ttl" directive:

```
sttl  <text for the subtitle>
```

The subtitle may be up to 52 characters in length. If the page option is not selected, this directive will be ignored. As with the "ttl" option, any number of "sttl" directives may appear in a source program. The subtitle can be disabled or turned off by an "sttl" command with no text following.

SYS

The 'sys' or system call directive allows the programmer to setup a system call.
Such a call consists of a TRAP#15 instruction followed by a two byte function code optionally followed by 32-bit parameter values.
This directive automatically inserts the TRAP, then obtains the function code and any other parameters from the operand field.

```
sys  <function>,<parameter1>,<parameter2>,...
```

The <function> and <parameter> values may be any legal absolute, relocatable or external expression. <function> will be stored as 16 bits, all <parameters> will be stored as 32-bits.

TTL

The 'ttl' directive allows the user to specify a title or name to the program being assembled. If the "pag" option is also selected, this title is then printed in the header at the top of each output listing page. If the page option is not selected, this directive is ignored. The proper form is:

```
ttl  <text for the title>
```

All the text following the 'ttl' directive (excluding leading spaces) is placed in the title buffer. Up to 32 characters are allowed, with any excess being ignored. It is possible to have any number of 'ttl' directives in a source The latest one encountered will always be the one used for printing at the top of the following page(s).

CONDITIONAL ASSEMBLY

The assembler supports conditional assembly -- the ability to assemble only certain portions of your source program depending on the conditions at assembly time. Conditional assembly is particularly useful in situations where you might need several versions of a program with only slight changes between versions.

As an example, suppose we required a different version of some program for four different systems whose output routines varied. Rather than prepare four different source files, we could prepare one that would assemble a different set of output routines depending on some variable which was set with an "equ" directive near the beginning of the source. Then it would only be necessary to change that one "equ" statement to produce any of the four final programs.

THE "IF-ENDIF" CLAUSE

In its simplest form, conditional assembly is performed with two directives: "if" and "endif". The two directives are placed in the source listing in that order with any number of lines of source between. The assembler evaluates the expression associated with the "if" statement (we will discuss this expression in a moment), and if the result is true, assembles all the lines between the "if" and "endif" and then continues assembling the lines after the "endif". If the result of the expression is false, the assembler will skip all lines between the "if" and "endif" and resume assembly of the lines after the "endif". The syntax of these directives is:

```
if  <expression>
  .
  .   conditional code goes here
  .
endif
```

The "endif" directive requires no additional information, but the "if" directive requires an expression. This expression is considered FALSE if the 32-bit result is equal to zero. If the result is not equal to zero, the expression is considered TRUE.

THE "IF-ELSE-ENDIF" CONSTRUCTION

An "else" directive may be placed between the "if" and "endif" statements. In effect, the lines of source between the "if" and "endif" are split into two groups by the "else" statement. Those lines before the "else" are assembled if the expression is true; those after (up to the "endif") are ignored. If the expression is false, the lines before the "else" are ignored while those after it are assembled. The "if-else-endif" construct appears as follows:

```
if  <expression>
 .
 .   this code is assembled if the expression is true
 .
else
 .
 .   this code is assembled if the expression is false
 .
endif
```

The "else" statement does not require an operand. There may be only one "else" between an "if-endif" pair.

It is possible to nest "if-endif" clauses (including "else"s). That is, an "if-endif" clause may be part of the lines of source found inside another "if-endif" clause. You must be careful, however, to terminate the inner clause before the outer.

Another form of the conditional directive, "ifn" ("if not") functions just like "if," except that the sense of the test is reversed. Thus, the code immediately following is assembled if the result of the expression is NOT TRUE. An "ifn-else-endif" clause appears as follows:

```
ifn  <expression>
 .
 .   this code is assembled if the expression is FALSE
 .
else
 .
 .   this code is assembled if the expression is TRUE
 .
endif
```

NOTE

In order for conditionals to function properly, they must evaluate to the same result in Pass One and Pass Two. Thus if labels are used in a conditional expression, they must be defined in the source before the conditional directive is encountered.

SPECIAL FEATURES

END OF ASSEMBLY INFORMATION

Upon termination of an assembly and before the symbol table is output, three items of information may be printed: the total number of errors encountered, the total number of excessive branches encountered, and the sizes of the text, data and bss segments.

The number of errors is printed in the following manner:

```
0 Errors detected.
```

Excessive branches (a long branch used where a short branch will suffice) are printed after the error count, for example:

```
1 Error detected.
3 Excessive branches detected.
```

The size of the segments are displayed as follows:

```
SEGMENT SIZES
TEXT SEGMENT = 00002C
DATA SEGMENT = 00010A
BSS SEGMENT  = 000006
```

All of this information may be suppressed by using the '+e' command line option; however, if errors are detected, this information will be displayed anyway.

EXCESSIVE BRANCH INDICATOR

To allow size and speed optimization of the final code, the assembler places a greater-than sign just before the address of any long branch instruction which could be replaced by a short branch. The total count is reflected in the end-of-assembly information previously described. The following section of code shows just how it looks:

```
000000                              text
000000 4A80                         tst.l   d0
>000002 6600 0006                    bne     lab1
000006 4A81                         tst.l   d1
000008 6602                         bne.s   lab2
00000A 2601             lab1        move.l  d1,d3
00000C 2800             lab2        move.l  d0,d4
00000E                              end
```

Note how the ".s" postfix was used to create a short branch.

AUTO FIELDING

The output assembly listing automatically places the four fields of a source line (label, mnemonic, operand, and comment) in columns. This allows the programmer to edit a condensed source file without impairing the readability of the assembled listing. The common method of doing this is to separate the fields by only one space when editing. The assembled output places all labels in column 25, all opcodes in column 34, and all operands in column 42 and comments start in column 56 unless the operands field extends into the comments. There are a few cases where this automatic fielding can break down (such as lines with errors), but these cases are rare and generally cause no problem. Labels that are longer than 8 characters are printed on a line by themselves (above the code they were with -- if any).

FIX MODE

Comment lines may begin with either an asterisk (*) or a semicolon (;). If a semicolon is used, the "+F" option of the assembler will assume that the "comment" is a valid instruction to be assembled. Therefore, the assembler will act as though the semicolon did not exist at all; the rest of the information on that line will be assembled. For example:

```
;label1 move.l #2,d0
; sys term
```

With the "+F" option invoked, these two lines will be normally assembled. This aides in the debugging process.

LOCAL LABELS

Local labels are available in the assembler. These local labels allow the programmer to reuse labels; in this way meaningless labels can be replaced with local labels. For more information on local labels, refer to the description of the label field in the Assembler Operation and Source Line Components discussion earlier in this section.

OBJECT CODE PRODUCTION

The object code output from the 4404 assembler is a standard 4404 relocating binary file for relocatable modules. This object code output can be turned on or off via the '+b' option on the calling line. The relocatable output module always requires processing by the linking loader to be executed. For more information about relocatable modules, refer to the discussion of the linking loader, later in this section.

RELOCATABLE (SEGMENTED) OBJECT CODE FILES

The 4404 operating system supports "segmentation" of binary files. It permits binary files to be broken into three segments of code: called "text", "data", and "bss." Each assembly module must contain one of these directives before any instructions that produce object code can be processed. The assembler does not default to any given segment when assembling a file.

Any code in a "text" segment is assumed by the operating system to be read-only. That is, it will only read code in a "text" segment and will not attempt to write into it.

The "data" segment is sometimes referred to as "initialized data". It is code which has been produced by the assembler and which can be either read or written. For example, the data segment might contain a temporary variable that requires an initial value. At any point, the variable could be read or re-written.

The "bss" segment is an area of reserved memory where no actual code has been produced by the assembler. It is sometimes referred to as "uninitialized data". The binary file does not contain any code to be placed in this section of memory, only a size value for this segment. Its main purpose is to tell the operating system that memory is required in this area, but it does not need to be initialized to any values. The "bss" segment or area of memory can be read or written.

Breaking the binary file into these three sections provides several benefits. The "text" segment is known to be read-only. This implies the code will never be altered as long as the program runs. The operating system can make use of this fact by sharing this segment of memory in the event that more than one users wish to run the program at the same time. This can mean a considerable increase in efficiency of the system. The "data" section must be different for each user running the program. It is information (actual instructions or data) which must be initialized or loaded, but which can be altered at some later point. The "bss" segment really contains no code or data in the binary file. It is just a signal to the operating system that when the file is loaded it needs memory allocated in the area specified. The program should not assume that the memory in this segment will be initialized to any particular value.

The assembler performs segmentation by maintaining three distinct location counters or program counters (PC's). At any point in the assembly, only one of these PC's is in effect. Any code generated by an instruction at that point is assembled at the address in the PC currently in effect. It is possible to switch to a different PC by use of one of the following three directives in the opcode field:

```
text
data
bss
```

It is necessary to state the segment that is desired before any executable code is produced. It is possible to change which segment code is currently being generated into at any time. In other words, you could begin with a "text" directive, enter 10 lines of code, then switch to the data segment with a "data" directive, enter 10 lines of code, then switch back to the text segment with another "text" directive, etc. To resume with the last address used by a particular segment, enter the segment directive:

```
text
move.l  10,d0
data
temp fcb   0
text
move.l  temp,a0
end
```

It is not possible to generate code in a "bss" segment. Any attempt to do so will result in an error.

Code generated into the "data" segment is actually written to a temporary file called '/tmp/asmbdataxxxxx' (xxxxx represents the current task number). At the end of the assembly, this data is copied onto the end of the text code found in the main output file, and the temporary file is immediately deleted.

THE BASE AND STRUCT DIRECTIVES

Two other directives related to PC's and segmentation are "base" and "struct." These directives are used almost exactly like a segment PC directive (especially the "bss" segment) but serve a different purpose. They are really just extra PC's which can be set and maintained for the purpose of establishing offsets from some fixed address in an area outside the three segments. Generally they are used in conjunction with storing information on a stack. Symbols declared in these segments can be absolute or relocatable, depending on the attributes of the operand. Symbols declared in a "struct" segment can be reused just as if they had been defined using the "set" directive. Symbols declared in a "base" segment may be used only once, like any other label. A short example program may be the best illustration:

```
stack equ     $EF0000

      base    $000000
temp  ds.w    1
saved ds.l    2
junk  ds.b    1

      text
start move.l   stack,a0
8     move.l  junk(a0),d2
      add.l   temp(a0),d2
      move.l  d2,saved(a0)
      bne.s   8b
      end
```

In this example, the "base" directive allowed us to set up the variables "temp", "saved", and "junk," which are offsets from the base location of the stack. Had a "struct" directive been used in place of the "base" directive, we could have reused the variables "junk", "temp" and "saved" in other stack structures. The "struct" directive is extremely useful in defining stack structures in subroutines, where names such as "ret_address", "frame_ptr", "arg1", etc., can be used over and over again without conflict. These directives do not actually create a segment, they merely set up a new PC which can be temporarily used to establish offset variables. These directives have most of the same permissions and restrictions as the "bss" segment; they default to location $000000 if first called without an operand. No code may be generated while the "base" or "struct" PC's are in effect, and new "base" or struct addresses are allowed. The segments end when a new segment begins.

GLOBAL

The "global" directive is used in relocatable modules to inform the assembler that the symbols declared global should be passed on to the linking loader. The syntax of the 'global' directive is:

```
global <label1>[,<label2>, . . .]
```

"label1", "label2", etc. represent the symbolic names of the labels to be declared as global; each label should be separated by a comma. The global directives must occur before the use or definition of the symbol. Normally, global symbols are declared at the beginning of the source module. Local labels cannot be declared global.

DEFINE AND ENDDEF

These convenience directives work much the way "global" works. The "define" directive informs the assembler that all labels declared in the label field will also be declared as global symbols. This "define" mode will be in effect until a "enddef" directive is encountered. For example,

```
data
define
temp1    fdb     0,$FFFF
start    move.l   1,d1
enddef
. . .
```

This example simply defines the two labels "temp1" and "start" as global. This directive works well when many symbols must be declared as global while they are initialized to various values.

EXTERN

The "extern" directive declares symbols to be external to this particular module. Local labels cannot be declared external. The syntax of the "extern" directive is:

```
extern <label1>[,<label2>, . . .]
```

"label1," "label2," etc. are ordinary labels as in "global;" labels should be separated by a comma. When the assembler encounters a label declared external in the operand field, external records will be written out to the binary output module. With the "global" directive, the "extern" directive should appear before the actual use of the external symbol, usually at the beginning of the source module. These external records will be used by the linking loader.

NAME

Each binary output module can be given a module name with the "name" directive. The module name is used by the linking loader in reporting errors and address information; it is strongly recommended to give each module a name. The syntax of the "name" directive is:

```
name <name of the module>
```

The module name can be a maximum of 14 characters. If more than one "name" directive occurs in the source module, the last name given will be the name assigned to the module.

COMMON AND ENDCOM

It is possible to establish common blocks in the assembler. These can only be named and uninitialized common blocks.

```
<name>  common
```

A common block declaration is terminated by the use of the "endcom" directive. The only directives allowed between the "common" and "endcom" are "rmb" and "ds," which define the size of the common block. Labels may be associated with each "rmb" or "ds" within the common block. For example:

```
test   common
temp1  ds.w     5
temp2  ds.l     10
       endcom
```

This common block is named "test," has two variables ("temp1" and "temp2") associated with it, and is 50 bytes long.

A common block and its variables are considered external by the assembler. Only one common block of a particular name should appear in a module. Because common blocks are treated as externals, the linking-loader handles the resolution of references to the common blocks automatically. For example:

```
       text
test   common
temp1  rmb    4
temp2  rmb    2
       endcom
start  move.l temp1,d0
       . . .
```

Common blocks are useful for passing parameters and keeping common information around. Furthermore, the common block will have the name of the common block as its module name; this is done automatically by the assembler. Common blocks must be accompanied by executable code in the same module that is, a common block cannot be the only item in a single source module.

ERROR AND WARNING MESSAGES

The assembler issues two types of error messages: fatal and non-fatal. A fatal error is one such as a disk file read error, which causes an immediate termination of the assembly. A non-fatal error results in an error message being inserted into the listing and some sort of default code being assembled if the error is in a code producing line. The assembly is allowed to continue on non-fatal errors. Error messages may not be suppressed.

All messages are output as English statements, not as error numbers. These messages announce violations of any of the rules and restrictions set forth in this manual and are essentially self-explanatory. Non-fatal error messages are preceded by with three asterisks, making them easy to locate.

Fatal error messages are sent to the standard error output. They are issued in the form:

```
Last Line = <last_line_read>
Line Number <line_num>
Fatal Error - <error_message_reported_here>
```

The messages which may come in the third line are listed later in this section. The last line processed is not reported on read, write, open or seek errors.

POSSIBLE NON-FATAL ERROR MESSAGES

16-bit expression expected.

A 16-bit expression was required in the operand field and the expression found cannot be represented in 16 bits.

8-bit expression expected.

An 8-bit expression was required in the operand field and the expression found cannot be represented in 8 bits.

A label declared 'global' was not found in the program.

All labels declared via the "global" directive must be defined in the module.

Absolute expression required.

An absolute expression is required in this context.

Branches not allowed across segment boundaries.

Branches cannot be made to labels in other segments or to externals.

Can't subtract two relocatables from different segments.

Subtraction of relocatables is not allowed if they are from different segments.

Couldn't evaluate expression.

The expression could not be evaluated.

Couldn't evaluate expression in pass1.

Assembler directives such as "ds" and "rmb" must be evaluated in both passes of the assembler. Only a constant operand is legal, and forward references are not allowed.

Couldn't find that local label.

The local label specified in the expression was not defined. Note that the local labels '0' and '00' are two distinct labels.

Data register required.

A data register is required as one of the operands for the instruction specified.

Duplicate label found.

The label on this line has been defined more than one time.

Evaluator : attempt to divide by zero.

The divisor of an expression evaluated to zero.

Evaluator : more than one external found in an expression.

Only one external variable can be used per expression.

Evaluator : must shift by positive, non-zero quantity.

Only non-negative shift amounts are legal.

Evaluator : not a valid operation for 2 reloc's or extern's.

The assembler detected an attempt to add to relocatables or externals. Only absolute expressions can be added to externals or relocatables.

Evaluator : operator only valid for absolutes.

The following operations can be performed on absolute expressions only: and, or, exclusive or, not, multiply, divide, shifts and the logical operators.

Evaluator : unbalanced expression (wrt segments).

The expression evaluator found an expression involving relocatables from different segments. In expressions containing relocatables, the relocatables must be paired and canceling. A relocatable expression can only be relocated relative to one segment.

Evaluator : unbalanced parenthesis.

The parentheses in the expression were not balanced properly.

External expression not allowed.

An external expression is not allowed in this context.

External symbol not allowed in this context

In some of the directives, an external symbol is not allowed. For example, the "equ" cannot have an external symbol in the operand field; a symbol cannot be equated to an external symbol.

Extra arguments found.

Only two operands were expected for this opcode, but more were found.

Found zero branch length on short branch.

A short branch cannot be made to the immediately following instruction.

Forced short but long expression found.

The expression which was forced short (via ":W") could not be fitted into a word.

IFDEF contained expression that couldn't be evaluated in Pass 1.

In conditional assembly, the assembler must be able to evaluate the condition in both passes. This expression can therefore not involve a forward reference to any variables.

Illegal addressing mode for instruction.

An addressing mode (specified in an operand) was is not legal for this instruction.

Illegal character in label.

Labels must consist only of alphabetic characters, digits and the underscore character.

Illegal expression or missing operand.

An expression could not be successfully parsed by the expression evaluator.

Illegal nesting of conditionals has occurred.

Conditional assembly rules have been broken. Conditionals can only be nested 20 levels deep and certain rules apply to their use. See the discussion of conditional assembly for a detailed description.

Illegal op-code for this segment.

Certain instructions cannot appear in some segments. For instance, no code can be generated in the BSS segment.

Illegal operand.

An error has been detected in the operand field.

Illegal register list for 'movem'.

The register list specified could not be interpreted. See the discussion of the instruction set for details on register list specification.

Illegal size for instruction.

The size specified by the .b/.w/.l extension is not allowed for this instruction.

Illegal special register for instruction.

The special register (USP,CCR,SR,VBR,...) specified as an operand is not legal for this instruction.

Immediate size does not match instruction size.

The immediate operand was larger than the size specified in the instruction, or implicit in the instruction.

Instruction expects only one operand.

The instruction specified has only one operand but more than one was found.

Invalid binary header flag.

The operand of the "bhdr" directive is not a known binary header flag. The legal binary header flags are:

Executable $04
Relocatable $05

Invalid local label - 0 thru 99 only.

Local labels must be in the range 0 through 99. Local labels may be reused in the same module.

Invalid option specified.

The only legal options to the "opt" directive are "con," "noc," "lis," "nol." See the discussion of pseudo-ops and directives for more details.

Invalid transfer address found (external).

External transfer addresses are not supported.

Label required.

The directives, set, equ and log require a label to be specified on the same line.

Negative value not allowed.

A negative value cannot be specified in this instruction.

Nested COMMON's not supported.

Common blocks cannot be nested.

No closing delimiter found.

The assembler found the EOL character before finding a closing delimiter in a string expression.

No ENDCOM directive found.

A common block declaration must be bracketed by the two directives "common" and "endcom". Another segment was entered without an "endcom" being specified.

Odd branch address found.

A branch to a label on an odd address was detected. Instructions must always begin on an even boundary. (The assembler takes care of this.) This can happen if a label is on a line by itself after some odd number of bytes of data, or the label is on a line with data that does not need to be aligned.

Overlapping register list specified.

The register list in the "MOVEM" instruction contains registers that have been specified more than once. The assembler issues this more as a warning than as an error, but the register list should be corrected.

Phasing Error.

The two passes of the assembler do not agree on the address of the label on the current line. This error can be caused by other errors in the assembly and should not appear as the only error in a given module. Only the first phasing error is reported, and checks are made only on lines containing labels.

Quicknumber (1-8) expected.

The instruction specified requires a "quick" count in the immediate operand field, and the expression found was not between 1 and 8 inclusive.

Relocatable displacement from the same segment required.

For PC relative code, the relocatable displacement must be from the same segment as the PC.

Relocatable displacement not allowed.

A relocatable displacement is not allowed in this context.

Relocatable expression required.

A relocatable expression is required in this context.

Symbol found in 'extern' also found as program label.

A symbol declared external to a module via the 'extern' directive cannot be defined in the same module.

The string was too long for the size specified.

The size specified in the instruction is smaller than the size of the immediate string specified as the first operand.

Too far for a branch instruction.

The target of a branch instruction must be within the constraints of a 16-bit expression. A jump will have to be used.

Too far for a short branch.

The target of a short branch must be within the constraints of an eight-bit expression. A long branch will have to be used here.

Undefined symbol found.

A symbol in an expression has not been defined.

Unknown addressing mode.

The addressing mode specified could not be interpreted by the assembler.

Unknown opcode.

The opcode on this line is not a known opcode. See the discussion of instructions and format for details.

Unknown size specified.

The only legal size extensions on instructions are 's', 'b', 'w', and 'l.' A size other than this was specified.

Word operand required on system call name.

The system call specified is not a legal system call.
The system call number must fit in 16 bits (word).
See Section 6, System Calls for more information.

POSSIBLE FATAL ERROR MESSAGES

Library file "<file_name>" not found

The specified library file could not be located. The assembler searches first in the current directory, then in a directory called "lib" in the current directory, and finally in the directory "/lib".

Library nesting too deep

Libraries may be nested only up to seven levels deep.

Local label table overflow

The maximum number of local labels allowed in a source file is 500.

No file specified

The assembler found no source files on the command line.

Opening "<file_name>": <reason>

The assembler received an error from 4404 while opening the specified file. An explanation of the error message is given.

Out of space

The assembler's symbol table is grown dynamically and grew to the limits of the size restrictions imposed by the 4404. The solution is to break the source into multiple modules and assemble them separately.

Reading "<file_name>": <reason>

The assembler received an error from the operating system while reading the specified file. An explanation of the error message is given.

Seeking in "<file_name>": <reason>

The assembler received an error from the operating system while seeking in the specified file. An explanation of the error message is given.

U requires label

The 'U' option requires a label as its argument. See the section on options for more details.

Unknown option '<char>'

The character specified is not a known option.

Writing to "<file_name>": <reason>

The assembler received an error from the operating system while writing to the specified file. An explanation of the error message is given.

THE LINKING LOADER

TERMINOLOGY

The remainder of this section describes the linking loader. The following additional terms are used:

loading — The placement of instructions and data into memory in preparation for execution. This preparation includes linking (the matching of symbolic references and definitions), and relocation of symbols and address expressions.

module — A subprogram which has been assembled using the assembler.

module name — The name given to a module by the programmer by using the "name" directive of the relocating assembler. If the "name" directive was not used, the module name is the same as the file name in which it is contained. Therefore, several modules may have the same name. The output module of the loader may be given a name by use of the "N" option.

relocatable object-code module — Equivalent to "module".

Linking Loader Input

The Linking Loader accepts independently assembled, relocatable object-code modules as input. Relocatable object-code is generated by the assembler "asm" in such a way that addresses are not bound to absolute locations at assembly time; this binding of the address fields will be accomplished by the Linking Loader. The "load" command binds the addresses at the time the object-code segments are combined to produce an executable program. The binding or adjustment of the address fields is termed " relocation". Relocation is necessary when an instruction expects an absolute address as an operand. The address field of this instruction must be increased by a "relocation constant". The relocation constant is the address at which the module is loaded for execution.

Address fields which do not require relocation are absolute addresses; their values remain the same regardless of the position of the object code segment in memory. Since the loader does not have access to the source text, it cannot determine if an address field is absolute or relocatable. In fact, it cannot distinguish addresses from data or opcodes. Therefore, the assembler must indicate to the loader which address fields require relocation. This communication is accomplished through "relocation records," which are appended to the object-code file produced by the assembler. Such a file is called a "relocatable object-code module."

Often it is desirable for parts of a program (called modules) to be developed separately. Each module must be assembled separately prior to final merging of all the modules. During this merging process, it is necessary to resolve references which refer to addresses or data defined in another module. The resolution of these "external references" is called linking. The assembler must provide information to the loader, in a manner similar to relocation records, concerning the address fields which must be resolved.

Linking Loader Output

As output, "load" produces an object-code module, a load map, a module map, and a global symbol table. The object-code module can be either relocatable or executable. A relocatable module produced by the loader cannot be distinguished from a relocatable module produced by the assembler. Only the loader, however, can transform multiple relocatable modules into an executable program.

The Standard Environment File

An environment file, "/ lib/std_env" is supplied with every 4404. The loader uses this file to get the information necessary to load a given module. The environment file is just an options file (described earlier in this section) which is processed before any other options on the command line. This file contains hardware-specific information so the user will not need to specify these things each time a file is loaded. Information such as the hardware page size and the starting address of the text or data segments is typically found here.

INVOKING THE LOADER

The linking loader accepts as input previously assembled, relocatable object-code modules and produces as output either:

1. A link-edited, relocatable, object-code module or
2. A link-edited, relocated, executable program.

The command line necessary to invoke the linking loader is:

```
++ load <relocatable_modules> [+options]
```

The two plus signs are the system's ready prompt, and "load" is the name of the linking loader command file.

"<relocatable_modules>" is a list of one or more file names, separated by blanks, which contain relocatable object-code modules you wish to load. The object-code modules will be loaded in the order specified.

"options" is a list of options which must start with a plus sign ("+") and may not contain any embedded spaces. More than one list of options may be specified, but each list must start with a plus sign. Some of the options are single characters, while others require an argument. Those that are single letters may be grouped together; for example: +sm. Those that require arguments may either stand alone or be the last of a group of options; for example: +smT=400000 (the "T=400000" is an option with an argument). The equal sign is not required in options with an argument. Therefore, "+T=400000" is equivalent to "+T400000".

Valid Options

+a=<minimum_number_of_pages>

The 'a' option specifies the minimum number of pages to be allocated to this task when executing. The number of pages specified must be a nonnegative decimal number.

+A=<maximum_number_of_pages>

The 'A' option specifies the maximum number of pages to be allocated to this task when executing. The number of pages specified must be a nonnegative decimal number, and should be greater than or equal to the minimum specified. The loader automatically adjusts "ridiculous" page counts.

+b=<maximum_logical_task_size>

The 'b' option tells the operating system of the largest size that this task may grow to while executing. The maximum logical task sizes currently supported are:

128K
512K
2048K (or 2M)
8192K (or 8M)

For example :

+b=512K
+b=2M
+b=8192K

The letters 'M' and 'K' may be in upper- or lowercase. The default task size is 512K. The loader automatically adjusts the task size if it finds that the size specified by the user or the default size is too small for the modules being loaded.

+c=<source_module_type>

The 'c' option allows the user to specify from what type of source file this module was created. This information is placed in the binary header for use in debugging. The source module types currently recognized are:

ASSEMBLER
C

and are specified as follows:

+c=ASSEMBLER
+c=C

Only uppercase letters may be used.

+D[=<start_of_data_segment>]

The 'D' option specifies the "data" segment bias to add to all "data" references (i.e., the starting address of the data segment). The number specified as the start of the data segment must be in hex and is machine dependent. If no starting data address is given, the data segment will follow the text segment. The 'D' option with no argument forces the data segment to follow the text segment (the default). This may be necessary if the "std_env" file contains data and/or text starting addresses and the user wishes to load a module for execution on another machine with different hardware requirements.

+e By default, the loader will notify the user only once about each unresolved external symbol. This option forces it to report every occurrence of every unresolved external symbol, showing in which module it was unresolved.

+F[=<options file name>]

This option allows the user to place loader command line options in a file rather than listing them each time on the command line. The file is read by the loader, and options are set from there as well as from the command line. The last occurrence of an option always overrides previous occurrences, so if the options file is specified first, any options found on the command line will override the same option in the options file. Nested options files are not supported.

The options specified in the options file must be separated by one or more spaces, may be on multiple lines, and must begin with a '+' just as they do on the command line. Only options may be specified in this file, not modules to be linked. The loader discards all characters up to a '+,' so comments may be inserted before the first option on a given line. For example, the following is a valid options file.

```
System XYZ Loader Options File
Data and Text Biases +D=400000 +T=0
Produce Maps +msM=mapout
Produce Shared Text +t
```

If the 'F' option is specified without an argument, the loader looks for the file called "ldr_opts" in the current directory, and uses it as the option file.

+i The "i" option includes all internal symbols in the symbol table for symbolic debugging. If the "i" option is not specified, only global symbols are included in the relocatable, object-code module.

+l=<library file name>

A maximum of five libraries may be specified by repeated use of the "l" option. If fewer than five libraries are specified, the system library is also searched in addition to the user libraries. Libraries are searched only when an executable output program is produced (not when 'r' is specified). In the following example, an effort is made to resolve externals not found in the user's modules by searching the three libraries "lib1," "lib2," and "Syslib:"

```
++ load echo[1-3].r +l=lib1 +l=lib2
```

For more information concerning the formation and use of libraries, see the discussion of libraries later in this section.

+L Do not search the libraries for unresolved externals.

+m Print the load map and the module map. The load map provides information as to the type of output file produced, the length of the resulting output object-code module, the number of input modules, and the transfer address. The module map describes the load address and object-code length for each input module.

+M=<map output file>

The load map, module map and symbol table are written by default to standard output. This option specifies a file name to which this information is to be written, rather than standard output.

+n Produces an executable output module with totally separate instruction and data spaces. This option informs the operating system that hardware support for separate instruction and data spaces is available and to handle addressing accordingly.

+N=<module name>

Specifies the name to be given to the output module of the loader (in a manner similar to the "name" directive of the assembler). Since the loader does not propagate the module names of the relocatable input modules to the output module, the "N" option must be used to assign a name to a module. If the "N" option is not used, the module name will default to the name of the file in which it is contained. Both executable programs and relocatable modules can receive module names. The name is limited to a maximum of 14 characters.

+o=<file name>

Specifies the file name for of the output binary file. If the "o" option is not specified, the output file will be named "file name.o" in the current directory. If a file by this name already exists, it will be deleted.

+r The "r" option specifies that the loader's binary output is to be a relocatable object-code module. The effect of this option is as if all the modules were contained in one source file and assembled with the assembler.

+s The "s" option directs the loader to print the global symbol table. If specified, the loader will print each global symbol and its address.

+S=<initial stack size>

This option informs the operating system that a task needs some amount of stack space when it begins execution. The operating system, by default, sets up a 4K-byte stack for each user task. This mechanism allows the task to begin execution with possibly more than the normal amount of stack space (but never less -- the operating system will always round up to the next 4K-byte boundary).

+t The "t" option specifies that the loader's binary output is to be a shared text, executable program. For more information, see "Shared Text Programs" in the discussion on segmentation.

+T=<start of text segment>

This option specifies the "text" segment bias to be applied to all text references. The starting address must be in hex; it defaults to 0 when the 'T' option is not specified. The text bias is machine dependent.

+u This option tells the loader not to print the "unresolved external" message when producing a relocatable output module.

LIBRARIES

Introduction

A library is a special collection of relocatable modules. When an external cannot be resolved from the user's modules, the libraries are searched in an effort to resolve it. The loader will search the user defined libraries in the order specified on the command line before searching the system library. This allows the user to redefine a system library module or entry point. The search for an external can be summarized as follows:

1. Can the external be resolved from the user's modules?

2. Can it be found in the user specified libraries?

3. Can the external be resolved from the system library?

When an external is resolved from a module contained in a library, that module is loaded and is then considered to be a "user" module. Because of this, library modules can reference other library modules.

The loader can search a maximum of five libraries when externals cannot be resolved from the user's modules. Usually, these libraries consist of up to four user libraries and the system library. The user can, however, specify five libraries on the command line. When five libraries are specified, the fifth one takes the place of the system library.

When searching for a library, the loader first looks for the specified file in the current directory. If the file is not found, the loader then looks for the "lib" directory in the current directory. If it finds that directory, the loader attempts to find the specified file. If not found, the loader makes a third and final attempt to find the specified file by looking in the directory "/lib". If the file is not found in any of these three directories, an error message is issued and the loader aborts. This process also is followed when searching for the system library.

Library Generation

The "libgen" utility is used to create new libraries and update existing libraries. All modules in a library **must** have a name. The name is assigned to a module by the "name" pseudo-op in the assembler or by the "N" option of the loader. It is the responsibility of the programmer to ensure that all modules in a library have names. The "libgen" utility will not accept a module without a name.

The syntax for the "libgen" utility is:

libgen o=<old>,n=<new>,u=<updates>,<options>,<deletions>

The arguments may be specified in any order.

The argument "o=<old>" specifies the name of an existing library file. This library file must have been created previously by "libgen". If "libgen" is begin called to create a new library (instead of updating an existing one), this argument should be omitted.

The argument "n=<new>" specifies the name of the new library. If this file already exists, it will be deleted before the new library is written. This argument need not be specified when updating an existing library. In this case, "libgen" will put the new library in a scratch file, delete the old library file, and rename the scratch file, giving it the name of the old library. The command line must include either the "o=<old>" or "n=<new>" argument, or both.

The argument "u=<updates>" specifies the name of a file containing modules that are to be added to the library, replacing existing modules in the library if necessary. More than one update file may be specified by repeating the "u=" argument. Up to nine files may be specified in this way.

As "libgen" runs, it produces a report, describing the action that it has taken for each module in the library. The report includes the module name and the file from which it was read (the old library or one of the update files). The options are used to eliminate or shorten this report. If the "+l" option is specified, no report will be produced. If the "+a" option is specified, the report will only contain information about those modules that were replaced, added, or deleted. No information about modules copied from the old library will be given.

The "<deletions>" argument is a list of module names to be deleted from the old library. The names may be separated by commas or spaces. If a name is specified that cannot be found in the old library, a warning message is issued. If the "+l" option was specified, no warning is issued.

EXAMPLES

1. Create a new library with the name "binlib" containing modules from the files "one", "two", and "three:"

   ```
   libgen n=binlib u=one u=two u=three
   ```

   Since a new library is being created, the "o=<old>" argument was omitted. Note that the "u=" argument was repeated for each update file.

2. Update the library named "binlib", adding or replacing records from the file "new." Produce an abbreviated report:

   ```
   libgen o=binlib u=new +a
   ```

   Since no new library was specified, the new library will be given the name of the old library.

3. Update the library named "binlib", deleting the modules named "diagonal" and "transpose;" add new modules from the file "xyz" and write the new library in the file "newlib:"

```
libgen obinlib u=xyz n=newlib transpose diagonal
```

SEGMENTATION AND MEMORY ASSIGNMENT

Relocatable and Executable Files

The loader can produce either an executable program or a relocatable module. By default, the loader produces an executable module: use of the 'r' option causes the loader to produce a relocatable module. The loader can produce two types of executable files: shared text and non-shared text. The next sections will discuss how "load" produces the relocatable modules and the two types of executable programs.

Relocatable Modules

Relocatable modules produced by the assembler have distinct text, data, and bss segments. All of the text object-code appears in the binary file first, followed by all of the data object-code. Since there is no object-code in the bss segment, it is thought of as following the data segment. The loader maintains these distinct segments by combining the text segments of all the relocatable input modules, followed by the concatenation of data segments, and then (conceptually) all the bss segments. In addition, the module segments are loaded in the order in which the modules are specified on the command line.

Common blocks (which contain only bss) are not combined with the bss segments of the other modules when producing a relocatable output module. Instead, common blocks retain their identity as separate modules and are appended to the resulting relocatable output module. Common areas will be combined with the bss segments of other modules only when producing an executable program.

Relocatable modules can be given module names by the use of the "name" directive of the assembler. This name is used when printing the module map. If no name was given to a module by use of the "name" directive, the name of the file in which it is contained is printed. When producing a relocatable output module, the loader does not propagate any of these module names to the output. To assign an output module a name, use the "N" option when invoking the loader.

Unlike module names, "info" fields are collected from the input modules and carried over to the relocatable output module and ultimately to the executable program.

Executable Programs

When loading modules to produce an executable program, the loader loads modules in the order specified on the command line. Common areas (which contain only uninitialized data) are loaded after the last module specified on the command line. Libraries are loaded after the last common block, or after the last user module on the command line if there are no common blocks.

The two types of executable programs the loader is capable of producing are shared text and non-shared text.

Shared Text Programs

Shared text programs have three distinct segments. The "text" segment is assumed to be read-only. This implies the code contained in the text segment will not be altered as long as the program runs. We can take full advantage of this fact by "sharing" this segment among several users who are running the program concurrently. This can mean a considerable increase in the efficiency of the system.

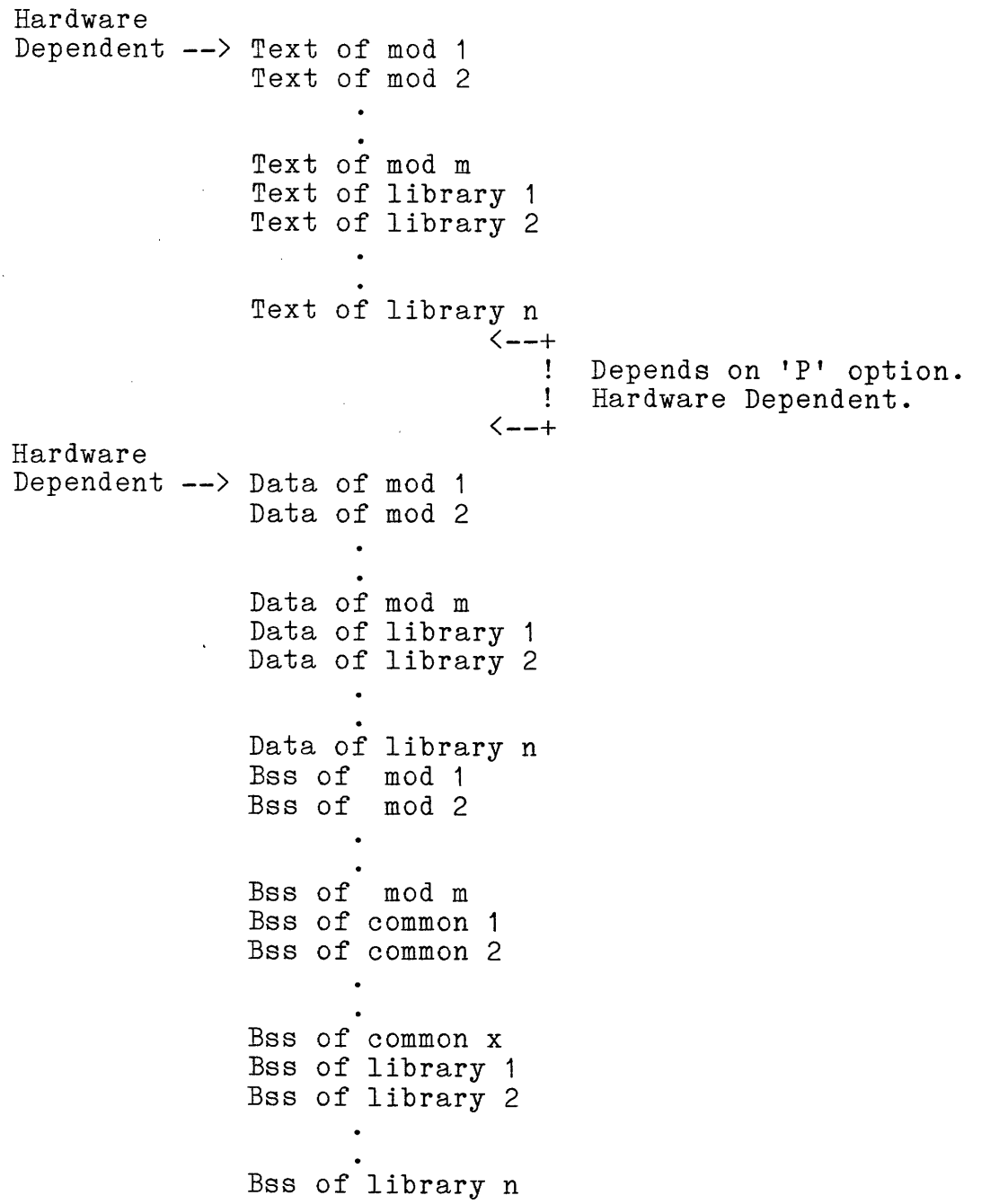
The "data" segment is also referred to as "initialized data". It is information (actual instructions or data) which must be initialized or loaded, but which can be altered at some later point. For example, counter variables which must be initialized to zero but will later be incremented should be placed in the data segment. At any time, the variable could be read or its value changed. Each user would then need his or her own copy of the data segment.

The "bss" segment, like the data segment, is also a read/write area. Since a module does not contain any object code to be loaded into this section of memory, it is also referred to as "uninitialized data". The module does contain the size of the bss segment, however, in order to inform the operating system that memory is required in this area but does not need to be initialized.

When producing a shared text program, the loader collects all the text segments from the relocatable input modules and loads them at the location specified by the 'T' option, or at 0 if no starting text address was given. All of the data segments are then placed either at the address specified in the 'D' option or, if no 'D' was given, immediately following the last text address, rounded up to the granularity specified in the 'P' option (or the next even byte if no 'P' was specified). Memory for the bss segments will be allocated immediately following the data segments at the time the program is executed.

There are drawbacks to using shared text. The text portion of a shared text file is always swapped to disk. Therefore, programs which are used infrequently, or those that only one task would be running at a time, would make better use of the system resources if they were non-shared.

The following memory map illustrates how the segments are loaded in relation to other segments and modules. The module numbers are the order in which they appear on the command line; "m" is the last module specified. Common blocks 1-x and library modules 1-n, which are loaded to complete the program, are also represented.


Hardware
Dependent --> Text of mod 1
Text of mod 2
Text of mod m
Text of library 1
Text of library 2
Text of library n
Depends on 'P' option.
Hardware Dependent.
Hardware
Dependent --> Data of mod 1
Data of mod 2
Data of mod m
Data of library 1
Data of library 2
Data of library n
Bss of mod 1
Bss of mod 2
Bss of mod m
Bss of common 1
Bss of common 2
Bss of common x
Bss of library 1
Bss of library 2
Bss of library n

Non-shared Text Programs

A non-shared text program has the same form as shared text program except that it is simply not shared. The non-shared text programs do not incur the overhead of having their text segments swapped immediately to disk at execution time. The memory map for a non-shared text program is the same as for a shared text program.

LOAD AND MODULE MAPS

Load Map

The 'm' option controls the printing of the module and load maps. The load map provides information about the type of output produced, the length of the resulting output object code module, the number of input modules, and the transfer address.

Module Map

Use of the 'm' option also selects printing of the module map. The module map describes the load addresses and object code length for each of the input modules.

The Module Map of a Relocatable Module

When producing a relocatable module, both the assembler and the loader do not "bind" or tie addresses to absolute locations; they are made relative to the base of the segment to which they refer. The following example assembled by the assembler will illustrate this point.

```
 1                                        extern  pdata
 2  000000                                text
 3 +000000 207C 0000 0000 Start          move.l  #msg1,a0
 4 X000006 4EB9 0000 0000                jsr     pdata
 5 +00000C 207C 0000 000A                move.l  #msg2,a0
 6 +000012 23C8 0000 0012 lab1           move.l  a0,msgaddr
 7 X000018 4EB9 0000 0000                jsr     pdata
 8  00001E 4E75                          rts
                      * Start of DATA segment
10  000000                               data
11  000000 4D65 7373 6167 msg1           fcc     "Message 1",0
12  00000A 4D65 7373 6167 msg2           fcc     "Message 2",0
                      * Start of BSS segment
14  000000                               bss
15  000000                               rmb     18
16  000012                    msgaddr    rmb     4
                      * Set transfer address
18          0000 0000                    end     Start
```

All of the segments start at address 0 (lines 2, 10, and 14). This is called the base address. Because of this, it is possible for two labels in different segments to have the same address (offset from the segment base). "lab1" and "msgaddr" are examples of this occurrence. All labels defined in a segment are relative to its base address. For example, "lab1" is 18 bytes from the beginning of the text segment, and "msg2" has an offset of 10 bytes from the base of the data segment. Throughout the linking process, the distance between "start" and "lab1" will remain constant. No assumptions, however, can be made about the distance between two labels that reside in different segments.

To produce a relocatable module from several input modules, the loader must combine all like segments. In other words, all text segments are concatenated starting with the text segment of the first input module, followed by the text of the second module, and so on. By doing so, however, the base address of all modules except the first will be changed. The loader automatically adjusts any addresses which refer to symbols in these modules which have been "relocated."

A small 'C' program was compiled, assembled and loaded, producing the following load and module maps:

```
* LOAD MAP *


Produced - executable, not overlapped TEXT and DATA.
Module is not shared text.
Starting TEXT address = 000000
Starting DATA address = 400000
Initial stack size = 000000
Granularity = 000000
Binary transfer address = 000B38
Number of input modules = 5


* MODULE MAP *


TEXT    DATA    BSS       MODULE NAME      FILE NAME
000000  400000  400204    test             test.r
000050  40000C  400204    Long Mul/Div     /lib/Clib
00032A  40000C  400204    C System Calls   /lib/Clib
000B04  4000A0  400204    strlen           /lib/Clib
000B38  4000A0  400204    C Wrapper        /lib/Clib


000B52  400204  400604    * Final Segment Addresses *
```

The maps show the text segments from each of the modules are combined and relocated to form the text segment of the final executable module. The starting text address was specified as 0. The starting data address was 400000. From looking at the module map we can see that all modules have text segments, the "Long Mul/Div" and the "strlen" modules have no data segments, and only the "C Wrapper" module has bss. Note also that the bss segment follows immediately after the data segment. The library "/lib/Clib" was searched successfully for the routines called directly and indirectly by the module "test". One can see that the binary transfer address is located in the "C Wrapper" module (Address $000B38).

The following map was produced using the same file as the previous map. No starting address for the data segment was specified, therefore the data segment follows the text segment. Since a granularity was specified as $1000 (The 'P' option), the data segment starts at the end of the text segment (rounded up to the next $1000 boundary). The executable module is to be shared text.

```
* LOAD MAP *


Produced - executable, not overlapped TEXT and DATA.
Module is shared text.
Starting TEXT address = 000000
Starting DATA address = 000000
Initial stack size = 000000
Granularity = 001000
Binary transfer address = 000B38
Number of input modules = 5


* MODULE MAP *


TEXT    DATA    BSS       MODULE NAME      FILE NAME
000000  001000  001204    test             test.r
000050  00100C  001204    Long Mul/Div R   /lib/Clib
00032A  00100C  001204    C System Calls   /lib/Clib
000B04  0010A0  001204    strlen           /lib/Clib
000B38  0010A0  001204    C Wrapper        /lib/Clib


000B52  001204  001604    * Final Segment Addresses *
```

MISCELLANEOUS

Transfer Address

A transfer address is the location at which execution is to start when the program is invoked. The 'end' directive in the relocating assembler can be used to indicate a transfer address.

Only one relocatable module to be included in a program should contain a transfer address. If more than one module has a transfer address, the loader prints an error message and aborts.

Resolution of Externals With Library Modules

The loader resolves externals in the following manner:

1. Combine all user modules.

2. Search libraries sequentially resolving all references that the user modules make to the library modules. (Primary references)

3. Search libraries again, this time resolving any external references made by the library modules brought in during step 2. (Secondary references)

When resolving externals with library modules, the loader always processes the libraries in the order specified on the command line. When resolving secondary references (step 3 above), if bringing in another library module introduces more unresolved externals, then the library search begins from the beginning again. This way, even though the same module may appear multiple times in different libraries, only the first occurrence of each module (as defined by the order of the libraries on the command line) is used.

ETEXT, EDATA, AND END

In certain applications, it is desirable to know the last location contained in a particular program segment (text, data, or bss). Due to the manner in which these modules are loaded, it would be very difficult to determine these locations in an applications program. To alleviate this difficulty, the loader has three global symbols which are always available and contain the location of the end of a segment. These three globals are ETEXT, EDATA, and END; they correspond to the ends of the text, data, and bss segments respectively.

ETEXT, EDATA,
and END may be used like any other user-defined global symbols. Since they behave like user-defined globals, they will always appear in the global symbol table listing. When used in a module, they should be defined as external. These special symbols are pre-defined, so users should not give these names to their own global symbols.

ERROR MESSAGES

The loader produces both fatal and non-fatal error messages. Fatal error messages are of the form:

```
Fatal Error: <description_of_error>
Loader aborted!
```

Non-fatal errors are produced in different forms for different messages.

Non-Fatal Error Messages

```
Warning: "/lib/std_env" not found.
```

The "/lib/std_env" file is supplied with every 4404. It is an options file which contains hardware-specific information so that the user does not need to enter them for each load. If you have not deleted or renamed the file purposely, you should contact your Tektronix service representative.

```
"<symbol_name>" unresolved in module "<module_name>".
```

The specified symbol was referenced in the specified module, but the symbol could not be located in any of the user supplied modules or in the libraries (if libraries are being searched). This may be expected if a relocatable file is being produced. If an executable file is being produced it is an error.

```
Symbol name clash: "<symbol_name>" in module module_name>".
```

The specified symbol has been globally declared in more than one module. The module specified is the one containing the second declaration of the symbol. The name of the global symbol will have to be changed in one of the modules, and the module will have to be reassembled.

```
Integer overflow in module "<module_name>".
Segment = <segment>.
Offset in module = <offset>.
```

When relocating a field in the module specified, the loader detected overflow out of the size field being adjusted. This may not always be an error. The address of the field relative to the specified segment is also reported. Subtracting from an external in a module can result in this message being produced when in fact the result of the subtraction is exactly as it should be. The user should look carefully at the code being loaded to determine if the error message should be ignored or not.

Two-Byte address overflow in module "<module_name>".
Segment = .
Offset in module = <offset>.

This error message is similar to the preceeding one, but with one slight difference. A two-byte address (absolute word addressing mode from the assembler) must be a positive, 16-bit expression to be a valid address, whereas the previous overflow message requires only that the result be an unsigned 16-bit expression. This message definitely indicates an error. An address was forced to absolute short in the assembler when it can not be.

Fatal Error Messages

Illegal minimum page allocation!

The minimum page allocation must be a positive integer. The number specified on the command line is illegal.

Illegal maximum page allocation!

The maximum page allocation must be a positive integer. The number specified on the command line is illegal.

Too many libraries!

A maximum of five libraries may be specified on the command line to the loader.

Nested 'F' options!

Option files cannot be nested. Multiple option files can be specified on the command line though.

Illegal configuration specified!

The configuration specified is not a known configuration. See the 'C' option for more information.

Illegal option '<char>'!

The character specified is not a known loader option. See the "options" discussion for more details.

Relocatable, but data/text start specified.
Conflicting options!

When producing a relocatable file as output, no starting text or data addresses can be given.

Opening "<file_name>": <reason>

The loader received an error from the operating system when it tried to open the specified file. An explanation of the error is given.

Reading "<file_name>": <reason>

The loader received an error from the operating system while trying to read the specified file. An explanation of the error is given.

Writing to "<file_name>": <reason>

The loader received an error from the operating system while trying to write to the specified file. An explanation of the error is given.

Seeking to <location> in "<file_name>": <reason>

The loader received an error from the operating system when it tried to seek to the specified location in the specified file. An explanation of the error is given.

Unknown module type!

The module type specified on the command line is not a legal type. The loader only recognizes "FORTRAN", "C", "PASCAL", "COBOL", and "ASSEMBLER". See the options discussion for more details.

Illegal task size!

The task size specified on the command line is illegal. Allowable task sizes are: 128K, 512K, 2048K, 8192K, 2M, or 8M. See the options discussion for more details.

No files given!

The loader found no files on the command line.

Illegal input file "<file_name>"!

The specified file is not a legal relocatable file produced by the assembler or the loader.

Library "<library_name>" not found!

The library specified could not be located in the current directory, a directory called "lib" in the current directory, or in the "/lib" directory.

Bad library format for "<library_name>"!

The library specified did not have the correct format for a library created by the "libgen" utility.

Multiple transfer addresses!

Only one module can contain a binary transfer address. The loader found two user-specified modules with transfer addresses.

Illegal relocation!

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

BSS instruction segment!

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

BSS transfer address!

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

Section 6

SYSTEM CALLS

INTRODUCTION

Sections 4 and 5 provided an introduction to the 4404 system calls and the use of "asm" and "load." This section describes each of the system calls, including errors that may be returned after the system call. This section is meant to be used with the assembler. If you want to make system calls from a high-level language, see the documentation for that language.

OVERVIEW

Assembly language programs on the 4404 interface to the operating system through system calls, perform functions such as file manipulation and task control. The calls are implemented with the TRAP #15 opcode followed by a one word function code which defines the call to be performed. Up to four 32-bit values (longs) may follow the function code, depending on the particular call. The 4404 assembler supports the "sys" pseudo-op which sets up the appropriate machine code for a system call. Its syntax is:

```
sys    function[,arg0,arg1,arg2,arg3]
```

where "function" is the system call number or name. This pseudo-op produces the TRAP code for the call, a single word for the function, and 32-bit values for each argument.

The arguments to system calls fall into three categories: numbers, pointers, and buffer addresses. Numbers may be bit patterns (as in the "chprm" call) or mode codes such as in "open". A 32-bit value is used, even if the number required will fit in 16 bits or less. Pointer arguments are used for calls which require a name or ASCII string (such as file names for "open" and "create"). The pointer is simply the address of the location of the string in memory. The string should always be null terminated (a 00 byte). A buffer address is used for calls, such as "status", that require a place in the caller"s address space to place data generated by the call. A buffer address is simply a 32-bit address pointing to the start of the data buffer to be used. Some calls also extract data from a caller-supplied buffer.

SECTION 6
System Calls

Some system calls require information to be passed in registers as well as through arguments. Most calls use the D0 register, but a few use A0 as well. All registers are preserved through a system call unless a value is returned in the register. An error generated in a call always returns the error number in the D0 register.

Condition codes are also preserved through a system call with the exception of the error bit. The error bit is the same as the carry, and the assembler supports the "bes" and "bec" mnemonics ("branch if error set" and "branch if error clear"). These mnemonics are synonymous with "bcs" and "bcc," respectively. The error bit will always return cleared if no error resulted from the call; otherwise it will be set and the error response code will be in D0. The usage of each system call will be described in a similar manner. To illustrate, here is an example of the "read" system call:

```
<file descriptor in D0>
sys read,buffer,count
<bytes read in D0>
```

The information in the angle brackets preceding the call shows the data the system expects to find in the registers. In this example, the D0 register should contain the file descriptor number of the file to be read. Next is the actual system call as it would appear in the assembler source listing. The system function is "read" and it has two arguments, "buffer" and "count". Following the call is information regarding the data to be found in the changed registers. In this example, the D0 register will contain a count which represents the actual number of bytes read from the specified file. Other registers will be unchanged.

NOTE

If a system call returns data to a buffer, it may not return it into the "text" segment of a program; it must be return the data to the "data" or "stack" segments. For example, the "buffer" in "read", and "tbuf" in "time" may not reside in the "text" segment.

NOTE

The "ind," system call signals an address error if the indirect target area is in the text segment. Keep the target area for indirect system calls in the data or stack segments.

SYSTEM ERRORS

When the system returns from a system call with the error (carry) bit set, register D0 contains the number of the resulting error. Here is a list of all system error numbers and their respective meanings:

1 EIO I/O error.

This can result from a CRC error, hardware malfunction, or defective media problem while reading or writing a device.

2 EFAULT System fault.

System faults are detected by the hardware and vary from system to system.

3 EDTOF Data section overflow.

This error can result from a "break" system call if the data section of a program is growing and overflows into the stack section.

4 ENDR Not a directory.

The file name specified is not a directory but the system call requires it to be one.

5 EDFUL Device full.

The device currently being written has no more available space.

6 ETMFL Too many files.

Each task is permitted a maximum of 16 open files at any one time.

7 EBADF Bad file.

The file descriptor given does not refer to an open file, or the file mode is not correct for the operation (e.g., the file is open for read and a write is attempted).

8 ENOFL No file.

The file name specified could not be found.

9 EMSDR Missing directory.

One of the directory elements specified in a pathname did not exist.

10 EPRM No permission.

An attempt was made to perform an action (such as file access) for which permission was denied.

11 EFLX File exists.

The system call requires the file to be previously non-existent.

12 EBARG Bad argument.

A bad argument was presented to a system call. This usually implies a number which is out of range or a non-existent mode code.

13 ESEEK Seek error.

An attempt was made to seek beyond the beginning of a file or beyond the physically possible maximum size of a file.

14 EXDEV Crossed devices.

An attempt was made to link to a file on a different device than the existing file.

15 ENBLK Not a block special file.

The file name specified was not a block special file, and the system call referenced requires it to be a block device (e.g. mount).

16 EBSY Device busy.

The device specified in an "unmount" is currently being used.

17 ENMNT File not mounted.

The file specified to an unmount call was not previously mounted.

18 EBDEV Bad device specified.

The system call requires a device type file as an argument.

19 EARGC Too many arguments.

Too many arguments were presented to an "exec" system call and the argument space overflowed. There is an upper limit of approximately 3000 bytes for arguments.

20 EISDR File is a directory.

The file specified is a directory, and the system call requires it to be a regular type file.

21 ENOTB File is not binary.

An attempt was made to execute a file that was not an executable binary file.

22 EBBIG Binary file too big.

The binary file specified to "exec" exceeds the physical address space limits.

23 ESTOF Stack overflow.

The stack space overflowed into the task"s data or text space.

24 ENCHD No children living.

A "wait" system call was executed with no living "child" tasks to wait for.

25 ETMTS Too many tasks active.

In attempting to fork a new task, the system exceeded its task limit. This error will also result if the system task table becomes full.

26 EBDCL Bad system call.

A system call function code was encountered that does not represent an existing system call.

27 EINTR Interrupted system call.

One of the program interrupts that the current task was catching occurred during the system call.

28 ENTSK No task found.

The task id referenced in the system call did not represent an active task in the system.

29 ENTTY Not a tty.

The system call ("ttyget" or "ttyset") requires the specified file to represent a tty type device.

30 EPIPE Write to broken pipe.

The system attempted to write data to a pipe that did not have an active read channel open.

31 ELOCK Record lock error.

The specified record can not be locked by this task. Another task has the requested record locked.

32 ETXOF Text segment overflow.

The program"s text segment has exceeded the original specified size.

33 EVFORK Illegal operation in "vforked" task.

See "vfork" for more details.

34 EDIRTY Mounted disk is dirty.

The disk you attempted to mount was not unmounted before system shutdown. Run diskrepair to clean up the disk.

SYSTEM DEFINITIONS

Several files containing system definitions reside in the "/lib" system directory. Use these files as "library" files in the assembler whenever the appropriate definitions are required. Here"s a general description of each file:

- o sysdef System call definitions. All of the system call names are defined in this file.
- o sysdisplay System display and event function code definitions. This file contains the information returned by the "getDisplayState" system call.
- o syserrors System errors. All standard system error names and their equated error numbers appear in this file.

- o sysstat — File status block. This file contains the block definition for the information returned by the "status" and "ofstat" system calls.
- o systim — Time buffer definitions. The "time" and "ttime" system calls return their information in a caller provided buffer. These buffers are defined in this file.
- o systty — TTY buffer for the console device. The "ttyget" and "ttyset" require a buffer for their data transferal. The contents of this buffer is defined here.
- o syscomm — TTY buffer for the communications device. Similar for "systty," but defined for the US-323C host communications port.
- o sysints — System program interrupts. All program interrupt names are equated to their respective numbers in this file.

DETAILS OF SYSTEM CALLS

alarm

Usage

```
<seconds in D0>
sys alarm
<previous seconds in D0>
```

Description

Alarm will cause an alarm interrupt to be issued after the number of seconds specified. At alarm time, the program interrupt SIGALRM will be sent to the task. Unless this interrupt is caught or ignored, it will terminate the task. This system call returns immediately to the caller after execution.

Diagnostics

No errors are possible from this call.

break

Usage

```
sys break,address
```

Description

Break changes the amount of memory associated with the task. The "address" specifies the highest address to be used by the task for data. If the address specified is already in the assigned data space, any memory beyond it will be released back to the system.

Diagnostics

An error is issued if more memory is requested than is physically possible on the system.

chacc

Usage

```
sys chacc,fname,perm
```

Description

Chacc checks the accessibility of file "fname". The "perm" argument should be "1" for read check, "2" for write check, or "4" for execute check. Any combination of these may be used (e.g. 3 checks read/write). If "perm" is 0, "chacc" checks if the directories leading to the file may be searched and if the file actually exists.

Diagnostics

Returns an error if the file does not exist, the directory path cannot be searched, or if the permission is not granted.

chdir

Usage

```
sys chdir,dirname
```

Description

"chdir" changes the current user directory to that specified by "dirname", which points to the actual name. The caller must have execute permission in the specified directory.

Diagnostics

Issues an error if the name specified is not a directory or cannot be searched.

chown

Usage

```
sys chown,fname,ownerid
```

Description

chown" changes the owner of the file name pointed at by "fname". Ownerid should have a maximum of 16-bit significance. Only the system manager may execute this call.

Diagnostics

Returns an error if the caller is not the system manager.

chprm

Usage

```
sys chprm,fname,perm
```

Description

chprm" changes the access permission bits associated with the file name pointed at by "fname". The new permission bits "perm" will replace the old. The allowable permissions are:

```
FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
FXSET => %01000000 ($40)  set id bit for execute
```

Diagnostics

Issues an error if the file does not exist, or the caller is not the file owner or system manager.

close

Usage

```
file descriptor in D0>
sys close
```

Description

close" closes the file represented by the specified file descriptor. Files are automatically closed when the task that opened them terminates, but it is wise to close them manually whenever possible.

Diagnostics

Returns an error if the file descriptor is not valid, or if the file has already been closed

cpint

Usage

```
sys cpint,interrupt,address
<old address in D0>
```

Description

cpint" tells the system what action it should take when "interrupt" occurs. If the specified address is 0, the default action will occur (usually task termination). If the address is 1, the interrupt will be ignored. An even address will be taken to be a valid user program address where control should be passed upon interrupt interception.

After interception, the interrupt number will be in the D0 register. The user"s code should exit the interrupt code via an RTR instruction. Following the return, the task will continue at the point it was interrupted.

After processing an intercepted interrupt, the system resets it back to the default condition; therefore, to continue catching the interrupt, it is necessary to re-issue a new "cpint" call each time the interrupt occurs. It should be noted that the SIGKILL interrupt cannot be ignored or caught. All interrupts retain their status after a "fork," but xec" resets all caught interrupts back to their default state. The system calls for "read" and "write" when referencing a slow device (like a terminal), and the calls "stop" and "wait" may return prematurely if a caught interrupt occurs during the system"s handling of

them. If this happens, it will look as if the system call returned an error (EINTR), and the call can be re-issued if desired.

In the following list of system interrupts, those marked with "*" cause a core dump if not caught or ignored.

SIGHUP	1	Hangup
SIGINT	2	Keyboard
SIGQUIT	3*	Quit
SIGEMT	4*	EMT $Axxx emulation
SIGKILL	5	Task kill
SIGPIPE	6	Broken pipe
SIGSWAP	7	Swap error
SIGTRACE	8	Trace
SIGTIME	9*	Time limit
SIGALRM	10	Alarm
SIGTERM	11	Task terminate
SIGTRAPV	12*	TRAPV instruction
SIGCHK	13*	CHK instruction
SIGEMT2	14*	EMT $Fxxx emulation
SIGTRAP1	15*	TRAP #1 instruction
SIGTRAP2	16*	TRAP #2 instruction
SIGTRAP3	17*	TRAP #3 instruction
SIGTRAP4	18*	TRAP #4 instruction
SIGTRAP5	19*	TRAP #5 instruction
SIGTRAP6	20*	TRAP #6-14 instruction
SIGPAR	21*	Parity error
SIGILL	22*	Illegal instruction
SIGDIV	23*	DIVIDE by 0
SIGPRIV	24*	Privileged instruction
SIGADDR	25*	Address error
SIGDEAD	26	Dead child
SIGWRIT	27*	Write to READ-ONLY memory
SIGEXEC	28*	Execute from STACK/DATA space
SIGBND	29*	Segmentation violation
SIGUSR1	30	User-defined interrupt #1
SIGUSR2	31	User-defined interrupt #2
SIGUSR3	32	User-defined interrupt #3

Diagnostics

Issues an error if the interrupt specified is out of range.

create

Usage

```
sys create,fname,perm
<file descriptor in D0>
```

Description

"create" creates a new file with the
access permissions specified in "perm".
The permissions are the same as in the "chprm" call, and are:

```
FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
```

If the file already exists, its length will be truncated to zero (all data deleted) but the original permissions and owner will be retained. In either case, the file is ultimately opened for writing. It is not necessary to specify write permission even though the file will ultimately be opened for writing. This allows a task to create a file and disallow others from writing the file until the task has been completed.

Diagnostics

Issues an error issued if too many files are open, if the files path can not be searched, or if the directory it resides in cannot be written.

crpipe

Usage

```
sys crpipe
<read file descriptor in D0>
<write file descriptor in A0>
```

Description

This call creates a pipe for inter-task communication. This call should be used before a "fork" operation, to allow the output of the original task to be used as input by the forked task. Up to 4096 bytes of output may be written into the pipe before the task will be suspended. Once the task doing the reading has read all of the data written, the writing task will again be run. If the writing task closes the file (file descriptor from A0) and the reading task consumes all of the data, an end-of-file condition will result.

Diagnostics

Issues an error if too many files are opened.

crtsd

Usage

```
sys crtsd,fname,desc,address
```

Description

This call creates a special file (device) or a new directory. "fname" specifies the name of the new file; "desc" is a 16-bit descriptor that describes the file"s type and permissions. If the file being created is a special file, the "address" argument specifies the internal device number. The descriptor has the "type" as the most significant byte and the "permissions" as the least significant byte. Their definitions follow:

Types

```
TPBLK => %00000010 ($02)  block type device
TPCHR => %00000100 ($04)  character type device
TPDIR => %00001000 ($08)  directory type file
```

Permissions

```
FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
FXSET => %01000000 ($40)  set id bit for execute
```

Diagnostics

issues an error if the file already exists or if the caller is not the system manager.

defacc

Usage

```
sys defacc,perm
```

Description

"defacc" set the default access permissions as specified by "perm". Normally, when a file is created, it is given the permissions specified in the "create" system call. The value specified by "create" is ANDed with the one"s-compliment of a per-task value known as the default permissions. This process will turn off or disable the permissions contained in the default permissions byte, no matter what the specified permissions are in the create call. The "defacc" call is used to set the default permissions. All "forks" and "execs" pass on the existing default value. See "chprm" for a list of the permission bits.

Diagnostics

No errors generated.

dup

Usage

```
<file descriptor in D0>
sys dup
<file descriptor in D0>
```

Description

"dup" duplicates the specified file descriptor; in other words, the file associated with the file descriptor is opened again and given another descriptor, which is returned. The new file is opened with the same mode as the original (e.g., if the original was open for "read", so will the new one).

Diagnostics

Issues an error if too many files are opened or the file descriptor is invalid.

dups

Usage

```
<file descriptor in D0>
<specified descriptor in A0>
sys dups
<file descriptor in D0>
```

Description

This call is like "dup" except the caller may specify the file descriptor of the duplicated open file. If the specified descriptor is already open, it is closed before being duplicated.

Diagnostics

Issues an error if too many files are open, or if the file descriptors are invalid.

exec

Usage

```
sys exec,fname,arglist
...
fname fcc "....",0
...
arglst fqb arg0,arg1,...,0
arg0 fcc "....",0
arg1 fcc "....",0
```

Description

The "exec" system call executes a binary file. "fname" specifies the file to be executed. The calling task will be terminated and the new one started up. There is no return from a successful exec. A return indicates an error condition. All open files remain open through the exec. Interrupts that are being ignored will stay in that state, but those that are being caught are reset to their default state.

When the file starts executing, the following arguments are available:

```
... highest address in task space  ...
0
...
arg0: <arg0 >
0
argn
...
arg0
sp -> argcnt
...   low memory ...
```

The stack pointer is pointing at a 4-byte argument count. Above that is a list of pointers to the actual arguments, which are at the highest part of memory. Two zero bytes are left at the very top of the task address space.

Diagnostics

Results in an error (and a return to the caller of exec) if the file does not exist, it was not executable binary, there were too many arguments (approximately 3,000 bytes max), or the memory space was exceeded.

filtim

Usage

```
<time in DO>
sys filtim,fname
```

Description

"filtim" sets the "last modified time" of the specified file to the value contained in the DO register. Only the system manager may execute this call.

Diagnostics

Returns an error if the file does not exist, if the file is currently open by another task, or if the caller is not the system manager.

fork

Usage

```
sys fork
<new task returns here>
<old task here (pc+2), new task id in DO>
```

Description

"fork" creates a new task. The new task inherits a copy of the caller's core image, all open files, and file pointers. The new task is identical to the original except that the old task returns 2 bytes past the system call and has the newly created task"s id in the DO register.

Diagnostics

Issues an error if too many tasks have been created or the system task table is full.

gtid

Usage

```
sys gtid
<task id in DO>
```

Description

This call returns the running task"s system id. This number may be used to generate unique file names.

Diagnostics

No errors are returned.

guid

Usage

```
sys guid
<actual user id in DO>
<effective user id in AO>
```

Description

"guid" returns both the actual user id (which identifies the person who actually logged on the system) and the effective id (which defines the current access permissions of the running task).

Diagnostics

No errors are possible.

ind

Usage

```
sys ind,label
```

Description

The "ind" system call is used where it is necessary to create system calls or their arguments on the fly (in the running program). The "label" points to an address that contains the actual call and its arguments. The task resumes execution after

the "sys ind" and not after the labeled code. Another "ind" or "indx" call may not be called from "ind."

Diagnostics

Issues an error if the value at the "label" is not a valid system call, or if it is an indirect call.

indx

Usage

```
sys indx
```

Description

This call is similar to "ind," but allows the system function code and arguments to be anywhere in memory, including the stack. Where "ind" had a label pointing to the system call and parameters, this call requires A0 to point to the call and parameters. One application of "indx" is to push the arguments and system call code on the stack, point to the call, then issue an "indx" call. Another "ind" or "indx"
call may not be called from "indx."

Diagnostics

Reports an error if the system function is not a valid system call, or if it is another indirect call.

link

Usage

```
sys link,fname1,fname2
```

Description

This call links "fname1" to "fname2". After the link, reference to "fname2" will access the contents of "fname1". The files contents and attributes are not changed in any way.

Diagnostics

Issues an error if "fname1" does not exist, if "fname2" already exists, if "fname2's" directory is write protected, if "fname1" is a directory, or if the file names are on different devices.

lock

Usage

```
sys lock,flag
```

Description

"lock" keeps a task from being swapped (that is, it locks a task in memory). Only the system manager may execute this call. If "flag" is non-zero, the task will be locked; if it is zero, the task will be unlocked.

Diagnostics

Issues an error if the caller is not the system manager.

lrec

Usage

```
<file descriptor in DO>
sys lrec,count
```

Description

"lrec" makes an entry in the system"s locked record table. Before the new entry is made, all other entries in the table associated with the calling task and the specified file will be removed. "count" represents the number of bytes in the file (record size) to be locked from the current file position. If the specified record overlaps any part of another task"s entry in the lock table for the same file, an error will result (ELOCK). Only regular files may be referenced (e.g., no devices, pipes, or directories). Closing a file will remove the lock table entry created as the "urec" system call will. Note that the part of the file specified is not actually "locked" from other's use, but proper use of the "lrec" and "urec" calls will have the same effect.

Diagnostics

Produces an error if there is no file for the specified descriptor, the file is not a regular file, the record is locked by another task, or the lock table is temporarily full.

memman

Usage

```
sys memman,function,start_address,end_address
```

Description

The "memman" system call is used to control regions of memory. The region of a task"s logical address space is specified by "start_address" and "end_address". The "function" argument defines the control activity:

Function	Operation
0	Clear the "dirty bit"
1	Lock the region in memory
2	Unlock the region from memory
3	Write disable the region
4	Write enable the region
5	Release the storage associated with the region

In all cases, the region operated on is a set of "pages" and may actually exceed the address range specified. As an example, if the range (hex) 1020 - 10a0 was specified, the region affected would be 1000 - 1fff.

Diagnostics

Issues an error if the function number is not valid, the address range specified is out of the task"s address space, or if the "start_address" is greater the the "end_address".

mount

Usage

```
sys mount,sname,fname,mode
```

Description

"mount" mounts a special file on the file system. The file "fname" should be a directory; after the mount, any reference to "fname" will reference the root directory of the special file (block device) "sname". The "mode" is normally 0; if it is non-zero, the device is mounted as "read only" (i.e. writing not permitted).

Diagnostics

Issues an error if "sname" is not an appropriate file, if it is already mounted, if "fname" does not exist, or if too many devices are currently mounted.

ofstat

Usage

```
<file descriptor in DO>
sys ofstat,buffer
```

Description

This call returns the status of an open file. The file is referenced by its file descriptor (obtained when the file was opened or created). The status information is returned in the user space pointed at by "buffer". See the "status" call for a description of the returned information.

Diagnostics

Returns an error if the file descriptor is not valid (i.e. the file is not open or the descriptor is out-of-range).

open

Usage

```
sys open,fname,mode
<file descriptor in DO>
```

Description

"open" opens an existing file. The file is opened for reading if "mode" is 0, for writing if "mode" is 1, or for both reading and writing if "mode" is 2. The file name opened is "fname". "open" returns a file descriptor that must be used for future file references.

Diagnostics

An error will be issued if the file does not exist, the path directories cannot be searched, too many files are open, or the permissions do not grant the requested mode.

phys

Usage

```
sys phys,object
<logical base address in D0>
```

Description

The "phys" system call permits access to certain system resources. Resources represented by "object" are:

Object	Resource
1	128K bit map
2	Shared 4K page 1
3	Shared 4K page 2
4	Time of day clock

The object numbers are defined above. If the number is positive, the resource will be mapped into the task"s address space. If the number is negative, it will be mapped out. An object number of 0 will unmap all previously mapped in resources. The logical address of the base of the mapped in resource will be returned in D0.

Diagnostics

Returns an error if the object number is not valid.

profile

Usage

```
sys profil,prpc,buffer,bsize,scale
```

Description

The "profile" call sets up a buffer and parameters that the system uses to profile a running task. If profiling is enabled, each time a clock tick occurs (every tenth second) a word in the "buffer" that corresponds to the current value of the program counter in the running task will be incremented. The "prpc" value represents the lowest address in the running task to be profiled. The argument "buffer" specifies the address of the profile buffer, and "bsize" specifies its size. The buffer size also determines the highest address in the running task to be profiled since pc addresses too large to be mapped into the buffer are ignored.

The "scale" value is used to scale the task program counter and must be a power of 2 (maximum size is 128). Profiling may be disabled by setting "scale" to 0 or 1.

Here's what happens when a clock interrupt occurs during execution of a task for which profiling is enabled:

1. The profile value "prpc" is subtracted from the task's current program counter, and the result is divided by the scale factor.

2. This value is then multiplied by 2 to form an offset into the "buffer".

3. If this offset is less than "bsize", the 16 bit word residing at "buffer"+"offset" is incremented by one.

Diagnostics

No errors are issued.

read

Usage

```
<file descriptor in D0>
sys read,buffer,count
<bytes read in D0>
```

Description

This call reads the file represented by the specified file descriptor. The memory in the user"s space pointed to by "buffer" is filled with data from the file. A maximum of "count" bytes will be read. All bytes requested will not necessarily be returned. If the file is a terminal, at most, one line will be returned. If the returned byte count is zero and no error is reported, the end-of-file has been reached.

Diagnostics

Issues an error if a physical i/o error occurred, or if a bad file descriptor or bad count was specified.

seek

Usage

```
<file descriptor in D0>
sys seek,position,type
<position in D0>
```

Description

"seek" positions a file's read/write pointer to the specified file location. The file is specified by the file descriptor. The argument "position" represents a four-byte, signed offset. The starting point for this offset is determined as follows by the "type" argument:

type	starting position
0	Position from the beginning of the file
1	Position from the current position
2	Position from the end of the file

The returned value is the resulting position of the file.

If a "seek" is performed past the end of the file when writing, a gap in the file will be created (no actual device space will be allocated). This gap will be read as zeros. To determine the current position in the file, use sys seek,0,1".

Diagnostics

Returns an error if a file descriptor is invalid or if the "seek" is attempted on a pipe.

setpr

Usage

```
<priority in D0>
sys setpr
```

Description

"setpr" sets the priority bias used by the system scheduler. The value specified is subtracted from the normal user priority, so the effect is that of lowering the task"s priority. Only the system manager may specify negative arguments (which will increase the task"s priority). The priority bias specified should be in the range of 25 to -25.

Diagnostics

No errors are issued.

spint

Usage

```
<task number in D0>
sys spint,interrupt
```

Description

This call sends a program interrupt to a task. The task is specified by its task number; the receiving task must have the same effective user id unless the caller is the system manager. The "interrupt" argument specifies which interrupt to send. See "cpint" for a list of interrupts.

If the specified task number is zero, the interrupt will be sent to all tasks associated with the caller"s control terminal. If the task number is -1 and the caller is the system manager, the interrupt is sent to all tasks in the system with the exception of tasks 0 and 1 (the scheduler and the initializer).

Diagnostics

Issues an error if the specified task does not exist or if the effective user id"s do not match.

stack

Usage

```
<address in A0>
sys stack
```

Description

The system will extend the user"s stack memory to include the address specified. If the address is higher than what is currently allocated, all lower memory will be released to the system. A task initially starts with stack space between 100 and 3000 bytes depending on the number of arguments passed from exec.

Diagnostics

Issues an error if the request for memory overflows into the data segment.

status

Usage

```
sys status,fname,buffer
```

Description

The file "fname" has its status read and returned to the user in the space specified by "buffer". The data returned by this call (as well as "ofstat") has the following format:

```
* buffer begin *

st_dev   rmb   2    device number
st_fdn   rmb   2    fdn number
st_fil   rmb   1    filler for alignment
st_mod   rmb   1    file modes - see below -
st_prm   rmb   1    permission bits - see below -
st_cnt   rmb   1    link count
st-own   rmb   2    file owner"s user id
st_siz   rmb   4    file size in bytes
st_mtm   rmb   4    last time file was modified
st_spr   rmb   4    future use only

* mode codes

FSBLK => %00000010  ($2)  block device
FSCHR => %00000100  ($4)  character device
FSDIR => %00001000  ($8)  directory

* permissions

FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
FXSET => %01000000 ($40)  set id bit for execute
```

Diagnostics

Issues an error if the file does not exist or the directory path cannot be searched.

stime

Usage

```
<time in DO>
sys stime
```

Description

This call sets
the system time and date. The time is measured in seconds from 0000 UTC January 1, 1980. Only the system manager may execute this call.

Diagnostics

Reports an error if the caller is not the system manager.

stop

Usage

```
sys stop
```

Usage

Stop halts a task until a program interrupt is received from "spint" or "alarm". When stop returns, it will always have an error (EINTR).

Diagnostics

See above.

suid

Usage

```
<user id in D0>
sys suid
```

Description

This call sets the effective and actual user id. This call may be executed only if the actual user id matches the id in the argument, or if the caller is the system manager.

Diagnostics

Issues an error if the caller is not the system manager or if the actual user id does not match.

term

Usage

```
<status in D0>
sys term
```

Description

"term" terminates a task. The status specified is made available to the parent task. The status is usually zero if there were no errors in the terminating task. A non-zero status should indicate some error condition. This system call does not return to the caller.

Diagnostics

No errors reported.

time

Usage

```
sys time,tbuf
```

Description

The "time" call returns the system's current time. Internally, the time is kept as a four-byte number, representing the number of seconds that have elapsed since 0000 January 1, 1980 UTC. The time information is placed at the address specified by "tbuf" and has the following format:

```
tm_sec  rmb  4   Time in seconds
tm_tik  rmb  1   Ticks in current second (tenths)
tm_dst  rmb  1   Daylight savings flag
tm_zon  rmb  2   Time zone
```

The "tm_tik" value may be used for finer measurements. The time zone word is the number of minutes of time westward from Greenwich (eastward would be a negative number). If "tm_dst" is non-zero, it implies that the local time zone should be altered for Daylight Savings during the appropriate part of the year.

NOTE

The "time" system call does not permit the result buffer to reside in the text segment. An attempt to do so results in an address error exception. Put the buffer in the data or stack segment.

Diagnostics

No errors are issued.

truncate

Usage

```
<file descriptor in D0>
sys truncate
```

Description

The truncate system call truncates an existing file's size. The file must be open for write and the file descriptor passed in D0. The file will be truncated at the current file

position. To truncate at a specified location, it is necessary to use the "seek" system call prior to truncate.

Diagnostics

Returns an error if the file descriptor is not valid or the file is not open for write.

ttime

Usage

```
sys ttime,buffer
```

Description

This call is used to obtain the accounting time information about a task. All times are represented in tenths of seconds. The information is returned to the user at "buffer" and has the following format:

```
ti_usr  rmb  4   Task's user time
ti_sys  rmb  4   Task's system time
ti_chu  rmb  4   Children's user time
ti_chs  rmb  4   Children's system time
```

The child times shown are the totals of all children tasks spawned by this task and its children.

Diagnostics

No errors are issued.

ttyget

Usage

```
<file descriptor in D0>
sys ttyget,ttbuf
```

Description

This call returns information about a terminal. The information returned is put in the 6-byte buffer pointed to by "ttbuf". The following formats describe the data:

```
* ttbuf *

tt_flg  rmb  1  Flags byte - see below -
tt_dly  rmb  1  Delay byte - see below -
tt_cnc  rmb  1  Line cancel char (default is ^X)
tt_bks  rmb  1  Backspace character (default is ^H)
tt_spd  rmb  1  Terminal speed - see below -
tt_spr  rmb  1  Stop output byte - see below -

* flags

RAW    => %00000001 ($01)  Raw i/o mode
ECHO   => %00000010 ($02)  Echo input characters
XTABS  => %00000100 ($04)  Expand tabs on output
LCASE  => %00001000 ($08)  Map upper->lower on input and vice versa
CRMOD  => %00010000 ($10)  Output cr and lf for cr
BSECH  => %00100000 ($20)  Echo backspace echo char
SCHR   => %01000000 ($40)  Single character input mode
CNTRL  => %10000000 ($80)  Ignore control characters mode

* delays

DELNL  => %00000011 ($03)  New line delay
DELCR  => %00001100 ($0C)  Carriage return delay
DELTB  => %00010000 ($10)  Tab delay
DELVT  => %00100000 ($20)  Vertical tab delay
DELFF  => %00100000 ($20)  Form feed (same as DELVT)

* speeds

INCHR  => %10000000 ($80)  Input ready to be consumed

* stop output

XANY   => %00100000 ($20)  Accept any character to restart output
XONXOF => %01000000 ($40)  Enable XON/XOFF for start/stop output
ESCOFF => %10000000 ($80)  Disable ESC for start/stop output
```

Diagnostics

Returns an error if the specified file is not a character device.

ttynum

Usage

```
sys ttynum
<terminal number in D0>
```

Description

This call returns the number of the calling task"s terminal. For example, "tty02" returns $0002 in the D0 register.

Diagnostics

No errors are issued.

ttyset

Usage

```
<file descriptor in D0>
sys ttyset,ttbuf
```

Description

This call sets the terminal information described in "ttyget". The data in "ttbuf" is exactly as described in "ttyget".

In normal use, you would first execute a "ttyget" system call to obtain the existing configuration. Next, use the logical operators AND or OR to set or clear the desired bits. (Be careful not to alter any bits other than those that must be changed.) Finally, execute the "ttyset" system call.

Diagnostics

Issues an error if the file specified is not a character device.

unlink

Usage

```
sys unlink,fname
```

Description

"unlink" removes the "fname" entry from a directory. If this is

the last link to the file, the file will be deleted and its device space will be freed. If the file is open, it will not be destroyed until the file is closed.

Diagnostics

Issues an error if the file does not exist, the directory cannot be written, or the directory path cannot be searched.

unmnt

Usage

```
sys unmnt,sname
```

Description

This call unmounts a special file "sname" from the system. The file associated with the special file reverts to its ordinary interpretation (see mount).

Diagnostics

Issues an error if the file system specified is busy or is not mounted.

update

Usage

```
sys update
```

Description

"update" updates all information on the disks; it writes out all data that is in memory waiting to be written to the disks.

Diagnostics

No errors are reported.

urec

Usage

```
<file descriptor in D0>
sys urec
```

Description

"urec" removes an entry in the system"s lock table (previously installed by "lrec"). All entries associated with the calling task and specified file are removed.

Diagnostics

Issues an error if the specified file descriptor is bad.

vfork

Usage

```
sys vfork
<new task returns here>
<old task here (pc+2), new id in D0>
```

Description

Vfork is a more efficient "fork" operation and is only available on virtual memory systems. Its operation is identical to fork but instead of the child task receiving new memory, it uses the same memory as the parent. After a vfork, the parent is halted until the child task either terminates or execs another file.

There are several restrictions placed on the child task created by vfork. The system will not let the child change its memory size or execute the system calls memman, fork, or vfork. The user of vfork should make sure the child task does not alter the stack frame in any way or change data that the parent is not expecting changed.

Diagnostics

Issues an error if too many tasks have been created or if the system task table is full.

wait

Usage

```
sys wait
<task id in D0>
<term status in A0>
```

Description

This call is used to wait for a program interrupt or the termination of a child task. A "wait" must be executed for each of a task's children. The task id of the terminated task is returned, as well as its termination status. The low byte of this status is the value passed by the "term" system call. A non-zero value here usually represents some sort of error condition. The high byte of the status is zero for normal termination. If non-zero, this byte will contain the interrupt number that caused it to terminate. If the most significant bit of the status is set, a core dump was produced as a result of termination. Consult "cpint" for a list of interrupt numbers.

Diagnostics

Issues an error if there are no children tasks.

write

Usage

```
<file descriptor in D0>
sys write,buffer,count
<byte count written in D0>
```

"write" writes "count" bytes of data from location "buffer" to the file specified by the file descriptor. If the returned byte count does not equal the requested count", it should be considered an error. Writes that are multiples of 512 bytes and begin on 512 byte address boundaries are the most efficient.

Diagnostics

Issues an error if the file descriptor is invalid or if a physical i/o error resulted.

Section 7

THE 4404 C COMPILER

INVOKING THE "C" COMPILER

OVERVIEW

The "C" compiler, invoked by the "cc" command, accepts as input "C" source files, assembly language source files and relocatable modules. "C" source files must end in ".c", assembly language source files must end in ".a" and relocatable modules must end in ".r". If assembly language source files are specified on the command line, then only relocatable files can be produced.

The compiler can produce as output intermediate language files, assembly language files, relocatable modules, an executable module, or a "C" source listing.

The "C" compiler is fully compatible with the System V "C" compiler.

SYNTAX

The syntax for invoking the "C" compiler is:

cc <file_name_list> [+acDfiIlLmMnNoOqrRstUvwx]

where <file_name_list> is a list of the names of the files to compile.

Options Available

a	Produce as output assembly language source files with a "a" extension.
c	Put comments in the assembly language file.
D<name>[=<defn>]	Command line "#define".
f	Produce an output module suitable for firmware.
I	Produce as output intermediate language files with a ".i" extension.
i=<dir_name>	Specify a directory for "#include" files.

l=<lib_name>	Specify a library name to be passed to the loader.
L	Produce a source listing and write it to standard output.
m	Produce load and module maps from the loader.
M	Leave the combined output as one ".r" file.
N	Produce a listing without expanding "#include" files.
n	Run the first pass only, do not produce any code.
O	Run the assembly language optimizer.
o=<file_name>	Specify the output file name.
q	Produce code that does calculations on "char" and "short" variables without first converting to "int".
R	Produce as output relocatable modules with a ".r" extension, and an executable module.
r	Produce as output relocatable modules with a ".r" extension.
s	Produce code that does not do stack growth checking.
t	Produce a shared-text, executable output module.
U	Produce a line-feed character ($0A) for ' n' rather than the default of carriage return ($0D).
v	Show each phase of the compilation process (verbose mode).
w	Warn about duplicate "#define" statements.
x=<ldr_option>	Pass the information following the '=' on to the loader for processing.

DETAILED DESCRIPTION OF OPTIONS

The 'a' Option

The 'a' option instructs the compiler to produce files containing assembly language source code as output. These files have the same name as the "C" source files on the command line except that the extension ".a" replaces the extension ".c". If more than one file is specified on the command line, then a ".a" file is produced for each file.

The 'c' Option

The 'c' option tells the compiler to insert comments into the assembly language file produced during the code generation phase of the compilation. The comments mark the beginning of each expression plus variable names and associated offsets. The 'c' option only makes sense when used in conjunction with the 'a' option.

The 'D' Option

The 'D' option allows the user to define variables on the command line as if they were defined in every one of the "C" source files by using the preprocessor command "#define". The syntax for this option is:

D=<name>[=<defn>]

where <name> is the name of a variable defined for the "C" preprocessor, to be replaced by <defn> in the source code. If no <defn> is provided, the value of the <name> is one. The definition is valid over all source files on the command line. If a source file does not wish to include this definition it can be "undefined" by using the preprocessor command "#undef". The variable is redefined though at the beginning of every source file. There is no limit to the number of 'D' options that can be specified on the command line.

The 'f' Option

The 'f' option instructs the compiler to produce code suitable for firmware use. The code produced by the compiler is placed in only two segments, TEXT and BSS. All code and strings are generated in the TEXT segment. All global variables are placed in the BSS segment. No initialized global variables are allowed since they would have to appear in the DATA segment.

The 'i' Option

The 'i' option allows the user to specify directories to be searched for "#include" files. The syntax for this option is

```
i=<dir_name>
```

where <dir_name> is the name of a directory to be searched. The search procedure for "#include" files is

1. Search in the directory of the source file.
2. Search in the current directory (exactly as specified).
3. Search in a directory called "include" in the current directory.
4. Search in the directory "/lib/include".
5. Search in the directory "/usr/include".

If the file name specified to the "#include" command is enclosed in "<>", the directory of the source file is not searched. If the file name specified begins with a '/' no searching takes place and the file name specified is used as the "#include" file.

The 'i' option specifies directories to be searched after the current directory but before the other "standard" places. There is no limit to the number of 'i' options on the command line.

The 'I' Option

The 'I' option instructs the compiler to produce files containing intermediate language as output. These files have the same name as the "C" source files on the command line except that the extension ".i" replaces the extension ".c". If more than one file is specified on the command line, then a ".i" file is produced for each file. The intermediate language file is not readable text.

The 'l' Option

The 'l' option allows the user to specify libraries to be searched by the loader for resolution of external references. The syntax for this option is

```
l=<lib_name>
```

where <lib_name> is the name of a library to be searched. The libraries are searched in the order specified on the command line and are searched before the standard library "/lib/clib". No more than eleven libraries should be specified on the command line.

The 'L' Option

The 'L' option instructs the compiler to produce a "C" source listing. "#include" files are included in the listing. The listing is sent to standard output and contains line numbers.

The 'm' Option

The 'm' option tells the compiler to obtain load and module maps from the linking loader, "load". The maps are explained in detail in Section 5, The Assembler and Linking Loader. The maps are sent to standard output.

The 'M' Option

The 'M' option instructs the compiler to compile, assemble and link, the source files specified on the command line. The output produced is one relocatable module. The name of the output module is the name specified with the 'o' option, or if none is specified, the name "output.r" is given to the file.

The 'n' Option

The 'n' option tells the compiler to perform only a syntax check of the "C" source code files specified on the command line. No code is generated.

The 'N' Option

The 'N' option instructs the compiler to produce a source code listing without expanding the "#include" files. The "#include" files are still read, but their contents are not shown in the source listing. The listing is sent to standard output and contains line numbers.

The 'o' Option

The 'o' option specifies the name of the file containing the executable module produced by the compiler. The syntax for this option is:

o=<file_name>

If the user does not specify the 'o' option, the name of the file can be one of two things: if only one file is specified on the command line, and the file is a "C" source file, the name of the file produced is the name of the source file without the ".c" extension. Otherwise the name of the file is "output" in the working directory. If a file by this name already exists, it is deleted without warning. The 'o' option can not be used in conjunction with the 'r' or 'a' options.

The 'O' Option

The 'O' option instructs the "cc" command to run the assembly langauge optimizer. The optimizer should not be run on hand written assembly language source files since it makes certain assumptions about the source files it reads, and it replaces the input file with its output file. The optimizer is not run on assembly language source files specified on the command line.

The 'q' Option

The 'q' option tells the compiler to generate code which does numeric and logical operations on variables of type "char" and "short". The "C" language requires that before any operations on variables of these types are performed, the variables must be converted to type "int". This conversion in most cases is totally useless and the resulting code is both larger and slower than it has to be. An example should make this clearer. For example:

```
ch1 = ch2 << 3;
```

where "ch1" and "ch2" are of type "char". The code resulting from this statement would be,

(1) Convert "ch2" to type "int". (sign extend)
(2) Shift the result of the conversion left 3 places.
(3) Convert the result of the shift to type "char".
(4) Assign the previous result to "ch1".

The conversions done in the code generated are superfluous. Overflow is ignored in "C", and the code generated without the conversions would have the exact same effect. The user should be careful though about places where the automatic conversion to "int" is assumed and is necessary for the correct results. If the statement above was actually,

```
in1 = ch2 << 3;
```

where "in1" is of type "int", then the 'q' option could cause the wrong code to be generated, depending on what was intended by the programmer and what the value in "ch2" is. If the 'q' option was specified, the variable "ch2" would not be converted to type "int" before the shift operation. Any overflow out of "ch2" would be lost. If the 'q' option was not specified, "ch2" would be converted to "int" before the shift, and any overflow would be retained for the assignment to "in1". An explicit cast of "ch2"

into type "int" would solve this problem. In practice, using the 'q' option does make the code smaller and faster but care should be taken in its use. It is recommended that a program be thoroughly debugged before attempting to use the 'q' option. After compiling with the 'q' option the program should once again be thoroughly checked out.

The 'r' Option

The 'r' option instructs the compiler to produce as output files containing relocatable modules. These files have the same name as the "C" source files on the command line except that the extension ".r" replaces the extension ".c". If more than one file is specified on the command line, then a ".r" file is produced for each file. This option may not be used in conjunction with the 'o' option.

The 'R' option

The 'R' option tells the compiler to produce as output files which contain relocatable modules and to produce an executable file. The relocatable files have the same name as the "C" source files specified on the command line except that the extension ".r" replaces the ".c" extension. The executable file's name is as described in the 'o' option.

The 's' Option

The compiler, by default, checks with the operating system to see if it needs to produce code which performs checks whenever it requires space on the stack. If not enough space is available, a runtime routine is called to grow the stack. The 's' option tells the compiler to produce stack checking code. This option is never needed for the 4404.

The 't' Option

The 't' option tells the compiler to produce a shared-text executable file as output. This option is merely passed on to the loader. Shared-text files are discussed in detail in Section 5, The Assembler and Linking Loader.

The 'U' Option

The 'U' option instructs the compiler to produce a linefeed character ($0A) for the "C" character constant ' n'. The default is to produce carriage-return ($0D).

The 'v' Option

The 'v' option tells the compiler to show each step of the compilation process. The "command line" for each step of the process is shown, with file name arguments and options sent to each command.

The 'w' Option

The 'w' option instructs the "C" preprocessor to warn the user about duplicate definitions of variables. Redefining a preprocessor variable is perfectly legal, but it can lead to some very difficult to find bugs.

The 'x' Option

The 'x' option is used to pass options directly to the linking loader, "load". The syntax for this option is,

```
x=<ldr_option>
```

where <ldr_option> is some valid option to the "load" command. No '+' is allowed in front of <ldr_option>. An example of its use is,

```
x=F=/lib/nonstd_env
```

where 'F' is a legal "load" option specifying an options file.

EXAMPLES

The following examples illustrate some of the uses of the "cc" command.

1. cc test.c
2. cc math.c float.c driver.c +o=testmath +Owsq
3. cc list.c +Ln
4. cc games.c help.c +D=DBG=1 +o=play +l=gamelib +t
5. cc prog.c +Nvqca

The first example compiles, assembles and links the file "test.c", producing as output the executable module "test".

The second example compiles, assembles and links the three files specified producing as output the executable module "mathtest". The assembly language optimizer is run, short and character operations are done without conversion to integers, and no stack checking code is produced. The compiler warns the user about duplicate definitions.

The third example compiles the file "list.c" but generates no code. A listing of the "C" source file is sent to standard output.

The fourth example compiles, assembles and links the file "games.c" and "help.c" producing as output the executable module "play". The variable "DBG" is defined on the command line and the library "gamelib" is searched before the standard libraries. The output module is shared text.

The fifth example compiles the file "prog.c" and produces as output the assembly langauge source file "prog.a". A listing is also produced and sent to standard output. The "#include" files are not included in the listing. Short and character operations are performed without conversion to integer and comments are included in the assembly language source file. Verbose mode is turned on.

LANGUAGE DESCRIPTION

The "C" compiler is fully compatible with Bell Laboratories System V "C" compiler. Advanced features such as " enumeration" types, passing, returning and assigning structures/unions and bit fields are supported. The types "unsigned char", "unsigned short", and "unsigned long" are all supported.

Object Sizes

Each variable defined in a "C" program requires some specific amount of space. The sizes of the basic types in bytes are:

Type	Size
char	1
short	2
int	4
long	4
"pointers"	4
float	4
double	8

The qualifier "unsigned" does not affect the size of the variable, and can be applied to variables of type "char", "short", "int", and "long". "Short" implies "short int", "long" implies "long int", and "unsigned" implies "unsigned int".

Register Variables

The storage class "register" may be applied to variables of all basic types except "float" and "double". Invalid "register" declarations are ignored and the storage class is made "auto". Up to four pointer variables and five data variables can be declared per function.

abort

Send a task-abort signal to the current task, causing the task to terminate immediately.

SYNOPSIS

```
void abort();
```

Arguments

None

Returns

Never

DESCRIPTION

This function sends a task-abort signal to the current task, which causes the task to terminate immediately. The task-abort signal can not be caught or ignored. The function never returns to the caller.

ERRORS REPORTED

None

NOTES

The termination status received by the parent of the current task contains an exit code of zero, a termination code indicating that the task terminated because of a task-abort signal, and a flag that indicates if a core-image file was produced.

SEE ALSO

System Call: signal(), wait()

Command: int

access

Check the accessibility of a file.

SYNOPSIS

```
include <errno.h>
int access(path, perms)
     char     *path;
     int       perms;
```

Arguments

path	The address of a character-string containing a pathname for the file whose accessibility the function is to check
perms	A value indicating the type of access to check

Returns

Zero if access is permitted, otherwise -1 with "errno" set to the system error code indicating the reason for denying access

DESCRIPTION

This function checks the accessibility of the file reached by the pathname in the character-string referenced by <path>. The value <perms> determines the type of access to be checked. The function returns zero as its result if the file exists and grants the requested access. Otherwise, it returns -1 with "errno" indicating the reason the access is denied.

The function returns -1 if the path could not be followed, a part of the path is not a directory, the pathname does not reach a file, or the file does not grant the effective user the requested access permissions.

The value <perms> is a bit-string which tells the function what types of access to check. It may be any combination of the following values:

```
0x01  Read
0x02  Write
0x04  Execute (search)
```

A zero <perms> value tells the function to check the path to the file and the existence of the file.

ERRORS REPORTED

EACCES The file's permissions do not grant the requested access type

EMSDR The path to the file could not be followed ENOENT The pathname does not reach a file ENOTDIR A part of the path is not a directory

NOTE

If the current effective user is the owner of the specified file, the function examines the permissions granted by the file for its owner to determine accessibility. Otherwise, it examines the permissions granted for users other than its owner.

SEE ALSO

System Call: chmod(), stat()

acct

Initiate or terminate system accounting.

SYNOPSIS

```
include <errno.h>
include <sys/acct.h>
int acct(path)
     char     *path;
```

Arguments

path	The address of a character-string containing a pathname for the file to which to write accounting records, or (char *) NULL

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

If <path> is not (char *) NULL, this function initiates system accounting. While system accounting is active, the system writes a system accounting record (described below) to the file reached by the pathname in the character-string referenced by <path> every time a task terminates. The referenced file must already exist. If <path> is (char *) NULL, the function terminates active system accounting, if any.

This function returns zero if it successfully performs its function, otherwise it returns -1 with "errno" set to the system error code. This function requires that the current effective user-ID be that of the system manager.

The function fails if <path> is not (char *) NULL and the path in the pathname can't be followed, a part of the path is not a directory, the pathname does not reach a file, or system accounting is already active. The function also fails if the current effective user is not the system manager.

The following structure describes the record written by the system to the specified file each time a task terminates.

```
struct acct
{
        short           ac_uid;
        long            ac_strt;
        long            ac_end;
        char            ac_syst[3];
        char            ac_usrt[3];
        unsigned int    ac_stat;
        char            ac_tty;
        char            ac_mem;
        unsigned int    ac_blks;
        char            ac_spare[2];
        char            ac_name[8];
};
```

The "ac_uid" entry contains the user-ID number associated with the task, "ac_strt" contains the system-time at the start of the task, "ac_end" contains the system-time at the end of the task, "ac_syst" (a three-byte integer) contains the number of CPU-seconds used by the system on behalf of the task, "ac_usrt" (a three-byte integer) contains the number of CPU-seconds used by the task, "ac_stat" contains task's termination status, "ac_tty" contains the task's controlling terminal number, "ac_mem" contains the maximum number of 1028-byte blocks of memory ever allocated to the task at one time, "ac_blks" contains the number of I/O units used by the task, "ac_spare" is currently unused, and "ac_name" contains the first eight characters of the command which initiated the task.

ERRORS REPORTED

EACCES	The current effective user is not the system manager
EEXIST	System accounting is already active
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

The function does not report an error if the <path> is (char *) NULL and system accounting is not currently active.

The operating system writes accounting records to the end of the specified file.

SEE ALSO

Command: /etc/sysact

alarm

Set the task's alarm clock.

SYNOPSIS

```
unsigned int alarm(sec)
     unsigned int   sec;
```

Arguments

sec	The number of seconds to elapse before sending an alarm signal to the current task

Returns

The number of seconds remaining from a previous alarm clock request or zero if none

DESCRIPTION

If <sec> is not zero, this function arms the task's alarm clock so that the system sends an alarm signal to the current task after <sec> seconds has elapsed. If the alarm clock was already armed, the function cancels the previous alarm clock request. If <sec> is zero, the function cancels the previous alarm clock request.

This function returns as its result the number of seconds remaining on a previous alarm clock request, or zero if there was no previous request.

ERRORS REPORTED

None

NOTES

An alarm signal causes the current task terminate unless it explicitly catches or ignores alarm signals.

The actual amount of time that elapses before the system sends the alarm signal may be slightly less than the requested time since the system tics occur on one-second intervals.

SEE ALSO

C Library: sleep()
System Call: pause(), signal(), wait()
Command: sleep

brk

Change the task's data segment memory allocation.

SYNOPSIS

```
include <errno.h>
int brk(addr)
     char     *addr;
```

Arguments

addr The requested end-of-segment address for the data segment

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the amount of memory allocated to the data segment so that the data segment's end-of-segment address is <addr>. If the function succeeds, it returns zero as its result. Otherwise, it returns -1 with "errno" set to the system error code describing the reason for the function's failure.

The function fails if the address <addr> is less than the lowest address in the data segment, or if it could not allocate enough memory to satisfy the request. If the requested end-of-segment address is higher than the data segment's current end-of-segment address, the function allocates memory to the segment. If the requested end-of-segment address is lower than the data segment's current end-of-segment address, the function releases memory from the segment.

ERRORS REPORTED

ENOMEM There is not enough memory available

NOTES

A segment's end-of-segment address is the lowest logical address that is higher than the highest logical address of memory allocated to the segment.

SEE ALSO

C Library: calloc(), EDATA, free(), malloc(), realloc()

System Call: cdata(), sbrk()

cdata

Change the task's data segment memory allocation.

SYNOPSIS

```
include <errno.h>
int cdata(addr)
     char     *addr;
```

Arguments

addr	The requested end-of-segment address for the data segment

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the amount of memory allocated to the data segment so that the data segment's end-of-segment address is <addr>. If the function allocates memory to the data segment, it allocates memory that is physically contiguous to the last page of memory allocated to that segment. If the function succeeds, it returns zero as its result. Otherwise, it returns -1 with "errno" set to the system error code describing the reason for the function's failure.

The function fails if the address <addr> is less than the lowest address in the data segment, or if it could not allocate enough contiguous memory to satisfy the request.

If the requested end-of-segment address is higher than the data segment's current end-of-segment address, the function allocates memory to the segment that is physically contiguous to the last page of the segment. If the requested end-of-segment address is lower than the data segment's current end-of-segment address, the function releases memory from the segment.

ERRORS REPORTED

ENOMEM There is not enough memory available

NOTES

A segment's end-of-segment address is the lowest logical address that is higher than the highest logical address of memory allocated to the segment.

On virtual memory systems, this function is functionally equivalent to the "brk()" function.

SEE ALSO

C Library: calloc(), EDATA, free(), malloc(), realloc()

System Call: brk(), sbrk()

chdir

Change the working directory.

SYNOPSIS

```
include <errno.h>
int chdir(path)
     char     *path;
```

Arguments

path	The address of a character-string containing a pathname to the directory to be the new working directory

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the working directory to the directory reached by the pathname in the character-string referenced by <path>. It returns zero as its result if it successfully changes the working directory to the specified directory. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the pathname could not be followed or a part of the pathname is not a directory.

ERRORS REPORTED

EMSDR	The path to the file could not be followed
ENOTDIR	A part of the path is not a directory or the file reached by the pathname is not a directory

SEE ALSO

C Library: getcwd()

Command: chd

chmod

Change a file's access permissions.

SYNOPSIS

```
include <errno.h>
include <sys/modes.h>
int chmod(path, perms)
     char     *path;
     int       perms;
```

Arguments

path	The address of a character-string containing a pathname to the file whose access permissions are to change
perms	A bit-string describing the permissions to set on the file

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the access permissions of the file reached by the pathname in the character-string referenced by <path> to those described by the bit-string <perms>. The function requires that the current effective user be the owner of the file or the system manager. The function returns zero as its result if it successfully changes the access permissions of the file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, a file in the path is not a directory, the pathname does not reach a file, or the current effective user is not the owner of the file or the system manager.

The value <perms> is a bit-string which describes the permissions to set on the file. The include-file "<sys/modes.h>" defines the following constants which describe the meanings of each bit used by the function in the bit-string:

```
S_IREAD   0x01
S_IWRITE  0x02
S_IEXEC   0x04
S_IOREAD  0x08
S_IOWRITE 0x10
S_IOEXEC  0x20
S_ISUID   0x40
```

The value S_IREAD grants reading permission to the owner of the file, S_WRITE grants writing permission to the owner, and S_IEXEC grants searching permission to the owner if the file is a directory, otherwise it grants execution permission. The value S_IOREAD grants reading permission to users other than the owner of the file, S_IOWRITE grants writing permission to others, and S_EXEC grants searching permission to others if the file is a directory, otherwise it grants execution permission. The value S_ISUID causes the effective user-ID to be changed to that of the owner of the file whenever the program contained in the file is executed. The results of the function are undefined if bits other than those defined above are set in the bit-string <perms>.

ERRORS REPORTED

EACCES	The current effective user is not the system manager or the owner of the file
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

SEE ALSO

System Call: chown(), fstat(), stat()

Command: ls, perms

chown

Change the owner-ID of a file.

SYNOPSIS

```
include <errno.h>
int chown(path, uid)
     char     *path;
     int       uid;
```

Arguments

path	the address of a character-string containing a pathname to the file whose owner-ID is to change
uid	The user-ID to be new owner-ID of the file

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the owner-ID of the file reached by the pathname in the character-string referenced by <path> to the value <uid>. The owner-ID of a file is the user-ID of the user which is the owner of the file. The function requires that the current effective user be the system manager. The function returns zero as its result if it successfully changes the owner-ID of the specified file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file which is not a directory, the path does not reach a file, or the current effective user is not the system manager.

ERRORS REPORTED

EACCES The current effective user is not the system manager

EMSDR The path to the file could not be followed

ENOENT The pathname does not reach a file

ENOTDIR A part of the path is not a directory

NOTES

The user-ID <uid> need not be found in the system password file.

SEE ALSO

System Call: chmod(), fstat(), stat()

Command: ls, owner

chtim

Change a file's modification date and time.

SYNOPSIS

```
include <errno.h>
int chtim(path, time)
     char     *path;
     long      time;
```

Arguments

path	The address of a character-string containing a pathname to the file whose modification date and time is to change
time	The system-time value to set as the file's modification date and time

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the modification date and time of the file reached by the pathname in the character-string referenced by <path> to the system-time value <time>. The function can't change the modification date and time of a file that is currently open by another task and it expects the current effective user to be the system manager. The function returns zero as its result if it successfully changes the specified file's modification date and time. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file that is not a directory, the path does not reach a file, the file is currently open by another task, or the current effective user is not the system-manager.

ERRORS REPORTED

EACCES	The effective current user is not the system manager
EBSY	The specified file is currently open by another task
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

The function does not demand the system-time value <time> be between the creation date of the file and the current time-of-day.

The system represents time as the number of seconds that has elapsed since the epoch. The system defines the epoch as 00:00 (midnight), January 1, 1980, Greenwich Mean Time.

Other functions which change a file's modification date and time are "chmod()", "chown()", "creat()", "link()", "open()", and " unlink()".

SEE ALSO

System Calls: chmod(), chown(), creat(), link(), open(), unlink()

Command: touch

close

Close an open file.

SYNOPSIS

```
include <errno.h>
int close(fildes)
     int  fildes;
```

Arguments

fildes	A file descriptor for the file to close

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function closes the file referenced by the file descriptor <fildes>. The function returns zero as its result if it successfully closes the file, otherwise it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor <fildes> is out of range or does not reference an open file.

ERRORS REPORTED

EBADF	The file descriptor is out of range or does not reference an open file

NOTES

The system automatically closes all files that a task has open when that task terminates.

SEE ALSO

C Library: fclose(), fopen()

System Call: creat(), dup(), dup2(), open(), pipe()

creat

Create a new file or truncate an existing file.

SYNOPSIS

```
include <errno.h>
include <sys/modes.h>
int creat(path, perms)
     char     *path;
     int       perms;
```

Arguments

path	The address of a character-string containing a pathname for the file to create or truncate
perms	A bit-string describing the access permissions to set on the created file

Returns

If successful, a file descriptor for the created or truncated file, otherwise -1 with "errno" set to the system error code

DESCRIPTION

If no file is reached by the pathname in the character-string referenced by the argument <path>, this function creates an empty file, assigns the current effective user-ID as the file's owner-ID, assigns the access permissions described by anding the bit-string <perms> with the one's complement of the current file-creation mask as the file's access permissions, and links the specified pathname to the file. It then opens the file for writing access ignoring the file's access permissions, setting the current file position to the beginning of the file.

If the pathname in the character-string referenced by <path> reaches a file, the function truncates the file so that its length is zero and opens the file for writing access, setting the current file position to the beginning of the file. It does not change the file's access permissions or file's owner-ID.

If the function succeeds, it returns as its result a file descriptor for the opened file. Otherwise, it returns -1 with "errno" set to the system error code. The function fails if the path could not be followed, the path contains a file that is not a directory, no more files can be created on the device to contain the file, or no more files can be opened by the task. The function also fails if the pathname does not reach a file and the directory reached by the path does not grant the current effective user writing permission, or the pathname reaches a file and that file doesn't grant the current effective user writing permission.

The value <perms> is a bit-string which describes the permissions to set on the file. The include-file "<sys/modes.h>" defines the following constants which describe the meanings of each bit used by the function in the bit-string:

```
S_IREAD   0x01
S_IWRITE  0x02
S_IEXEC   0x04
S_IOREAD  0x08
S_IOWRITE 0x10
S_IOEXEC  0x20
S_ISUID   0x40
```

The value S_IREAD grants reading permission to the owner of the file, S_WRITE grants writing permission to the owner, and S_IEXEC grants searching permission to the owner if the file is a directory, otherwise it grants execution permission. The value S_IOREAD grants reading permission to users other than the owner of the file, S_IOWRITE grants writing permission to others, and S_EXEC grants searching permission to others if the file is a directory, otherwise it grants execution permission. The value S_ISUID causes the effective user-ID to be changed to that of the owner of the file whenever the program contained in the file is executed. The results of the function are undefined if bits other than those defined above are set in the bit-string <perms>.

ERRORS REPORTED

EACCES	The existing file or the directory to contain the link to the new file does not grant the user writing permission
EMFILE	The maximum number of files are open
EMSDR	The path to the file could not be followed
ENOSPC	The are no available file description nodes on the device which was to contain the specified file
ENOTDIR	A part of the path is not a directory

NOTES

This function opens the created file for writing even if the access permissions assigned to the file do not grant writing permission to the current effective user

If the task has the maximum number of files open and the specified file doesn't exist, this function creates the file, but does not open it.

SEE ALSO

C Library: fcreat(), fopen()

System Call: chmod(), chown(), open(), umask()

Command: create

dup

Duplicate a file descriptor.

SYNOPSIS

```
include <errno.h>
int dup(fildes)
     int       fildes;
```

Arguments

fildes	The file descriptor to duplicate

Returns

If successful, the duplicate file descriptor, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function duplicates the file descriptor <fildes>. The effect is that of opening again the file referenced by <fildes>, using the same open-mode and positioning to the current file position. If the function successfully duplicates the file descriptor <fildes>, it returns the duplicate file descriptor. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the task can't open any more files or the file descriptor is out of range or does not reference an open file.

ERRORS REPORTED

EBADF	The file descriptor is out of range or does not reference an open file
EMFILE	The maximum number of files are open

NOTES

The function always uses the lowest numbered available file descriptor.

SEE ALSO

System Call: close(), creat(), dup2(), open(), pipe()

dup2

Duplicate a file descriptor onto a specific file descriptor.

SYNOPSIS

```
include <errno.h>
int dup(src, dest)
     int       src;
     int       dest;
```

Arguments

src	The file descriptor to duplicate
dest	The target file descriptor

Returns

If successful, the duplicate file descriptor <dest>, otherwise -1 with "errno" set to the system error code

DESCRIPTION

If the file descriptor <dest> references an open file, this function closes that file. The function then duplicates the file descriptor <src> onto the specified file descriptor <dest>. The effect is that of opening again the file referenced by <src>, using the same open-mode and positioning to the current file position. If the function successfully duplicates the file descriptor <src> onto the file descriptor <dest>, it returns the file descriptor <dest>. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if either of the file descriptors are out of range or the file descriptor <src> does not reference an open file.

ERRORS REPORTED

EBADF	One or both of the file descriptors are out of range or the file descriptor <src> does not reference an open file

NOTES

If the file descriptors <src> and <dest> are the same file descriptor, the function returns <dest> without checking either file descriptor for validity.

If the file descriptor <dest> references an open file, the function doesn't close that file if the function fails.

SEE ALSO

System Call: close(), creat(), dup(), open(), pipe()

execl

Execute a program found in an executable binary file.

SYNOPSIS

```
include <errno.h>
int execl(path, [arg0, [arg1, ...,[ argn,]]] nullp)
     char     *path;
     char     *arg0, *arg1, ..., argn;
     char     *nullp;
```

Arguments

path	The address of the character-string containing a pathname for the file containing the program to execute
arg0	The address of the character-string containing the argument to the new program which is referenced as argument zero (by convention this is the name of the command)
arg1	The address of the character-string containing the first argument to the new program
argn	The address of the character-string containing the last argument to the new program
nullp	A null-address ((char *) NULL) which ends the list of addresses of character-strings containing arguments to the new program

Returns

Never if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function requests that the operating system replace the program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by <path>, that it pass as arguments to the new program the character-strings referenced by the values passed to this function following the argument <path> through but not including the argument <nullp>, if any, and that it begin executing the new program at its transfer address.

When the new program begins, it inherits the following attributes and resources from the calling program:

The task's priority
The task-ID number
The parent task-ID number
The user-ID number
The controlling terminal number
The file-creation permissions-mask
The time remaining on an armed alarm-clock
The working directory
All open files
System and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set, in which case the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the new program's signal-handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the new program's stack so that the number of arguments, represented by an int, is on the top of the stack, followed by addresses of character-strings which contain copies of those character-strings referenced by the arguments of this function, followed by the null-address ((char *) NULL), followed by the copies of the character-strings referenced by the arguments of this function.

This function only returns to the caller if the operating system reports an error. If it reports an error, the function returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the file's access permissions do not grant the current effective user execution permission. It also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

ERRORS REPORTED

E2BIG	Too many arguments are specified
EACCES	The file's permissions do not grant the requested access type
EBBIG	The executable file is too large
EISDR	The file is a directory
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOEXEC	This file is not executable
ENOTDIR	A part of the path is not a directory

NOTES

The function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All programs written in a high level language for this system expect their arguments in the form described above. For example, all C programs set up their stack so that the argument count is referenceable by the first argument to the main procedure "main()" and the address of the list of addresses of arguments is referenceable by the second argument to that procedure.

SEE ALSO

C Library: system()

System Call: execlp(), execv(), execvp(), fork(), profil(), signal(), vfork()

Command: shell, script

execlp

Execute a program found in an executable binary file.

SYNOPSIS

```
include <errno.h>
int execlp(path, [arg0, [arg1, ...,[ argn,]]] nullp)
char     *path;
char     *arg0, *arg1, ..., argn;
char     *nullp;
```

Arguments

path	The address of the character-string containing a pathname for the file containing the program to execute
arg0	The address of the character-string containing the argument to the new program which is referenced as argument zero (by convention this is the name of the command)
arg1	The address of the character-string containing the first argument to the new program
argn	The address of the character-string containing the last argument to the new program
nullp	A null-address ((char *) NULL) which ends the list of addresses of character-strings containing arguments to the new program

Returns

Never if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function requests that the operating system replace the program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by <path>, that it pass as arguments to the new program the character-strings referenced by the values passed to this function following the argument <path> through but not including the argument <nullp>, if any, and that it begin executing the new program at its transfer address.

When the new program begins, it inherits the following attributes and resources from the calling program:

The task's priority
The task-ID number
The parent task-ID number
The user-ID number
The controlling terminal number
The file-creation permissions-mask
The time remaining on an armed alarm-clock
The working directory
All open files
System and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set, in which case the new program gets as its effective user-ID that of the owner-ID of the file. The operating system sets up the new program's signal-handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the new program's stack so that the number of arguments, represented by an int, is on the top of the stack, followed by addresses of character-strings which contain copies of those character-strings referenced by the arguments of this function, followed by the null-address ((char *) NULL), followed by the copies of the character-strings referenced by the arguments of this function.

This function only returns to the caller if the operating system reports an error. If it reports an error, the function returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the file's access permissions do not grant the current effective user execution permission. It also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

ERRORS REPORTED

E2BIG Too many arguments are specified

EACCES The file's permissions do not grant the requested access type

EBBIG The executable file is too large

EISDR The file is a directory

EMSDR The path to the file could not be followed

ENOENT The pathname does not reach a file

ENOEXEC This file is not executable

ENOTDIR A part of the path is not a directory

NOTES

The function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All programs written in a high level language for this system expect their arguments in the form described above. For example, all C programs set up their stack so that the argument count is referenceable by the first argument to the main procedure "main()" and the address of the list of addresses of arguments is referenceable by the second argument to that procedure.

This function is exactly like "execl()" and is included only for compatibility with other systems. On other systems, this function follows the current search rules to locate the file containing the program to execute. This system doesn't support the notion of search rules except in the context of the shell, so the complete behavior of this function can't be fully implemented.

SEE ALSO

C Library: system()

System Call: execl(), execv(), execvp(), fork(), profil(), signal(), vfork()

Command: shell

execv

Execute a program found in an executable binary file.

SYNOPSIS

```
include <errno.h>
int execv(path, argv)
     char     *path;
     char     *argv[];
```

Arguments

path	The address of a character-string containing a pathname for the file containing the program to execute
argv	The address of a list of addresses of to character strings to pass as arguments to the new program

Returns

Never if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function requests that the operating system replace program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by <path>, that it pass as arguments to the new program the character-strings referenced by the addresses in the array referenced by <argv>, if there are any, and that it begin executing the new program at its transfer address.

The argument <argv> is the address of an array of (char *) containing the addresses of the character-strings which are arguments to new program, followed by the null-address (char *) NULL.

When the new program begins, it inherits the following attributes and resources from the calling program:

The task's priority
The task-ID number
The parent task-ID number
The user-ID number
The controlling terminal number
The file-creation permissions-mask
The time remaining on an armed alarm-clock
The working directory
All open files
System and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set, in which case the new program gets as its effective user-ID the owner-ID of the file. The operating system sets up the new program's signal handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the new program's stack so that the number of arguments, represented by an int, is on the top of the stack, followed by addresses of character-strings which contain copies of those character-strings referenced by the values in the list of references, followed by the null-address ((char *) NULL), followed by the copies of the character-strings referenced by the addresses in the array referenced by <argv>.

This function only returns to the caller if the operating system reports an error. If it reports an error, the function returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the file's access permissions do not grant the current effective user execution permission. It also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

ERRORS REPORTED

E2BIG	Too many arguments are specified
EACCES	The file's permissions do not grant the requested access type
EBBIG	The executable file is too large
EISDR	The file is a directory
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOEXEC	This file is not executable
ENOTDIR	A part of the path is not a directory

NOTES

The function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All programs written in a high level language for this system expect their arguments in the form described above. For example, all C programs set up their stack so that the argument count is referenceable by the first argument to the main procedure "main()" and the address of the list of addresses of arguments is referenceable by the second argument to that procedure.

SEE ALSO

C Library: system()

System Call: execl(), execlp(), execvp(), fork(), profil(), signal(), vfork()

Command: shell, script

execvp

Execute a program found in an executable binary file.

SYNOPSIS

```
#include <errno.h>
int execvp(path, argv)
     char     *path;
     char     *argv[];
```

Arguments

path	The address of a character-string containing a pathname for the file containing the program to execute
argv	The address of a list of addresses of to character strings to pass as arguments to the new program

Returns

Never if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function requests that the operating system replace program currently executing with the program found in the executable binary file (the new program) reached by the pathname in the character-string referenced by <path>, that it pass as arguments to the new program the character-strings referenced by the addresses in the array referenced by <argv>, if there are any, and that it begin executing the new program at its transfer address.

The argument <argv> is the address of an array of (char *) containing the addresses of the character-strings which are arguments to new program, followed by the null-address (char *) NULL.

When the new program begins, it inherits the following attributes and resources from the calling program:

The task's priority
The task-ID number
The parent task-ID number
The user-ID number
The controlling terminal number
The file-creation permissions-mask
The time remaining on an armed alarm-clock
The working directory
All open files
System and user time information

The new program inherits the effective user-ID unless the file has the set-user-ID mode-bit set, in which case the new program gets as its effective user-ID the owner-ID of the file. The operating system sets up the new program's signal handling mechanism like that of the calling program, except that all signals caught by the calling program are set up so that they cause their default action. The operating system disables profiling in the new program.

The operating system sets up the new program's stack so that the number of arguments, represented by an int, is on the top of the stack, followed by addresses of character-strings which contain copies of those character-strings referenced by the values in the list of references, followed by the null-address ((char *) NULL), followed by the copies of the character-strings referenced by the addresses in the array referenced by <argv>.

This function only returns to the caller if the operating system reports an error. If it reports an error, the function returns -1 with "errno" set to the system error code.

The function fails if the path could not be followed, the path contains a file that is not a directory, the pathname does not reach a file, the file is a directory, or the file's access permissions do not grant the current effective user execution permission. It also fails if the arguments to the program take up too much space (the maximum is system dependent but is always at least 2048 bytes) or the program in the file is too large.

ERRORS REPORTED

E2BIG	Too many arguments are specified
EACCES	The file's permissions do not grant the requested access type
EBBIG	The executable file is too large
EISDR	The file is a directory
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOEXEC	This file is not executable
ENOTDIR	A part of the path is not a directory

NOTES

The function does not flush or close standard I/O streams opened in the calling program before requesting that the new program be executed. All buffered data is lost.

All programs written in a high level language for this system expect their arguments in the form described above. For example, all C programs set up their stack so that the argument count is referenceable by the first argument to the main procedure "main()" and the address of the list of addresses of arguments is referenceable by the second argument to that procedure.

This function is exactly like "execv()" and is included only for compatibility with other systems. On other systems, this function follows the current search rules to locate the file containing the program to execute. This system doesn't support the notion of search rules except in the context of the shell, so the complete behavior of this function can't be fully implemented.

SEE ALSO

C Library: system()

System Call: execl(), execlp(), execv(), fork(), profil(), signal(), vfork()

Command: shell, script

fork

Create a new task.

SYNOPSIS

```
include <errno.h>
int fork()
```

Arguments

None

Returns

If successful, the child's task-ID to the parent (calling) task and zero to the child (created) task, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function creates a new task (the child task) that is an exact copy of the current task (the parent task). If the function succeeds, it returns the child task's task-ID to the parent task and returns zero to the child task. Otherwise, it returns -1 with "errno" set to the system error code.

The child task is identical to the parent task in that it has the same task priority, user-ID, effective user-ID, controlling terminal information, file-creation permissions-mask, working directory, signal handling set-up, and profiling information.

The child task differs from the parent task in that its task-ID is different, its parent task-ID is the task-ID of the parent task, the data in its memory is an exact copy of that in the parent task's memory, its file descriptors are exact copies of those in the parent task, and its system and user CPU times are reset to zero.

ERRORS REPORTED

EAGAIN	The maximum number of tasks for the user are active or there are no available entries in the system task table
EVFORK	The task shares its memory with its parent and may not call this function

NOTES

A task-ID is a non-negative integer.

Flushing or closing streams opened for write or append access at the "fork()" call may result in data being duplicated onto the file attached to the stream since buffers are copied to the child task by "fork()".

SEE ALSO

C Library: exit(), _exit()

System Call: execl(), execlp(), execv(), execvp(), vfork(), wait()

fstat

Get the status of an open file.

SYNOPSIS

```
include <errno.h>
include <sys/stat.h>
int fstat(fildes, bufad)
     int            fildes;
     struct stat   *bufad;
```

Arguments

fildes	A file descriptor for the open file to examine
bufad	The address of the structure to contain the file's status

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function examines the file referenced by the file descriptor <fildes> and writes information describing the status of that file into the structure whose address is <bufad>. The function returns zero as its result if it successfully gets the status of the open file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor <fildes> is out of range or does not reference an open file.

The following structure is defined in the include-file "<sys/stat.h>" and defines the format of the data describing the status of the open file:

```
struct stat

{
     short     st_dev;
     short     st_ino;
     char      st_filler;
     char      st_mode;
     char      st_perm;
     char      st_nlink;
     short     st_uid;
     long      st_size;
     long      st_mtime;
     long      st_spr;
};
```

The value "st_dev" is the device number of the device containing the file; "st_ino" is the file descriptor number (FDN) on the device describing the file; "st_filler" is an unused byte; "st_mode" is a bit string describing the type of the file, described below; "st_perm" is a bit-string describing the permissions of the file, described below; "st_nlink" is the number of links to the file; "st_uid" is the owner-ID of the file; "st_size" is the size of the file, in bytes; "st_mtime" is the last modification date and time for the file, in system-time; and "st_spr" is unused.

The following constants, defined in the include-files "<sys/stat.h>" and "<sys/modes.h>", define the data in the bit-string "st_mode" which describes the file's type:

```
S_IFMT    0x4F
S_IFREG   0x01
S_IFBLK   0x03
S_IFCHR   0x05
S_IFDIR   0x09
S_IFPIPE  0x41
```

The constant S_IFMT is a mask that when anded with the value "st_mode" yields the file type. After anding with the constant S_IFMT "st_mode" produces S_IFREG if the file is a regular file, S_IFBLK if the file is a block-special file (block device), S_IFCHR if the file is a character-special file (character device), S_IFDIR if the file is a directory, or S_IFPIPE if the file is a pipe.

The following constants, also defined in the include-files "<sys/stat.h>" and "<sys/modes.h>", define the data in the bit-string "st_perms" which describes the file's access permissions:

```
S_IREAD   0x01
S_IWRITE  0x02
S_IEXEC   0x04
S_IOREAD  0x08
S_IOWRITE 0x10
S_IOEXEC  0x20
S_ISUID   0x40
```

The value S_IREAD grants reading permission to the owner of the file, S_WRITE grants writing permission to the owner, and S_IEXEC grants searching permission to the owner if the file is a directory, otherwise it grants execution permission. The value S_IOREAD grants reading permission to users other than the owner of the file, S_IOWRITE grants writing permission to others, and S_EXEC grants searching permission to others if the file is a directory, otherwise it grants execution permission. The value S_ISUID causes the effective user-ID to be changed to that of the owner of the file whenever the program contained in the file is executed.

ERRORS REPORTED

EBADF	The file descriptor is out of range or does not reference an open file

NOTES

The include-file "<sys/modes.h>" need not be included if the includefile "<sys/stat.h>" since "<sys/stat.h>" includes "<sys/modes.h>".

SEE ALSO

System Call: creat(), dup(), dup2(), open(), pipe(), stat(), utime()

Command: ls

ftime

Get the operating system's current time statistics.

SYNOPSIS

```
#include <sys/timeb.h>
int ftime(tbufaddr)
     struct timeb  *tbufaddr;
```

Arguments

tbufaddr The address of a structure to get the operating system's current time information

Returns

Zero

DESCRIPTION

This function writes the operating system's current time statistics into the structure whose address is <tbufaddr>. The function always returns zero as its result.

The following structure definition describes the data written to the structure whose address is <tbufaddr>:

```
struct timeb
{
     long      time;
     char      tm_tik;
     char      dstflag;
     short     timezone;
};
```

The value "time" is the current system-time; "tm_tim" is the number of ticks (tenths of a second) that have passed since the last change in the system-time; "dstflag" is non-zero if converting to time coordinates of the local time zone requires the U. S. A. Standard Daylight Savings Time conversion, zero otherwise; and "timezone", if positive, is the number of seconds the local time zone is west of Greenwich Mean Time (GMT), otherwise its absolute value is the number of seconds east of GMT. The include-file "<sys/timeb>" contains definitions defining this structure.

ERRORS REPORTED

None

NOTES

The system represents time in system-time, which is the number of seconds that has elapsed since the epoch. The system defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time (GMT).

SEE ALSO

C Library: gmtime(), localtime(), tzset()

System Call: stime(), time()

Command: date, udate

geteuid

Get the current task's effective user-ID number.

SYNOPSIS

```
int geteuid()
```

Arguments

None

Returns

The current task's effective user-ID number

DESCRIPTION

This function gets the current task's effective user-ID number and returns that value as its result.

ERRORS REPORTED

None

SEE ALSO

System Call: getuid(), setuid()

Command: who

getpid

Get the current task's task-ID number.

SYNOPSIS

```
int getpid()
```

Arguments

None

Returns

The current task's task-ID number

DESCRIPTION

This function gets the current task's task-ID number and returns that value as its result.

ERRORS REPORTED

None

SEE ALSO

System Call: exec(), fork()

Command: status

getuid

Get the current task's user-ID number.

SYNOPSIS

```
int getuid()
```

Arguments

None

Returns

The current task's user-ID number

DESCRIPTION

This function gets the current task's user-ID number and returns that value as its result.

ERRORS REPORTED

None

SEE ALSO

System Call: geteuid(), setuid()

Command: who

gtty

Get the characteristics of an open character-device.

SYNOPSIS

```
#include <errno.h>
#include <sys/sgtty.h>
int gtty(fildes, buf)
     int            fildes;
     struct sgttyb *buf;
```

Arguments

fildes	The file descriptor of the open character-device to examine
buf	The address of the structure to contain the characteristics of the character-device

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function obtains the current characteristics of the open character-device referenced by the file descriptor <fildes> and writes information describing those characteristics into the structure referenced by <buf>. The function returns zero as its result if it successfully obtains the characteristics of the open character-device. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor is out of range, does not reference an open file, or does not reference an open character-device.

The function call expects <buf> to be the address of a structure which is defined as follows:

```
struct sgttyb
{
     unsigned char  sg_flag;
     unsigned char  sg_delay;
     unsigned char  sg_kill;
     unsigned char  sg_erase;
     unsigned char  sg_speed;
     unsigned char  sg_prot;

};
```

The bit-string "sg_flag" describes the current mode of the terminal. The values in the bit-string are as follows:

RAW	0x01
ECHO	0x02
XTABS	0x04
LCASE	0x08
CRMOD	0x10
SCOPE	0x20
CBREAK	0x40
CNTRL	0x80

If RAW is set, the operating system considers the character-device to be in raw mode. In raw mode, the operating system suspends all processing of input and output. If clear, the operating system considers the character-device to be in non-raw mode (sometimes called "cooked mode"). In this mode, the operating system processes characters dependent upon the setting of the other bits in the bit-string.

If ECHO is set, the operating system echoes characters read to the character-device. If clear, the operating system doesn't echo characters to the device. If XTABS is set, the operating system expands tab characters to spaces during output operations so that the next character written to the device is written to a column number that is an even multiple of eight. If clear, the operating system writes tab-characters to the character-device with no expansion. Tab-characters are defined by the system to be 0x09 and are defined by the C compiler as ' t'.

If LCASE is set, the operating system changes all upper-case characters to lower-case characters during input operations and changes all lowercase characters to upper-case characters during output operations. If clear, the operating system disables this feature. If CRMOD is set, the operating system writes a line-feed character to the character-device after every carriage-return character written. If clear, the operating system disables this feature.

If SCOPE is set, the operating system writes a backspace-character (0x08) followed by a space-character followed by another backspace characters to the character-device whenever a character-cancel character is read from the character-device. If clear, the operating system disables this feature. If CBREAK is set, the operating system considers the terminal to be in single-character mode. In this mode, the operating system reads data from the device one character at a time, passing each character to the calling task. If clear, the operating system considers the terminal to be in line mode, where it reads data from the device one line at a time, passing data to the calling task whenever a terminator is read.

If CNTRL is set, the operating system ignores all characters read from the character-device that are outside of the range 0x20 through 0x7E inclusive, except for the line-terminator character (carriage return), the keyboard-interrupt character (control-'c'), the quit-interrupt character (control-' '), the character-cancel character, the line-cancel character, and the output-stop and output-start characters if any.

The bit-mask "sg_delay" indicates which characters, if written to the character-device, causes the operating system to pause before writing another character to the character-device. The values in that bit string are as follows:

DELNL	0x03
DELCR	0x0C
DELTB	0x10
DELVT	0x20
DELFF	0x20

If DELNL is set, the operating system pauses it writes a new-line character to the character-device. If DELCR is set, it pauses after a carriage-return character, if DELTB is set, it pauses after a horizontal-tab character, if DELFF is set (which is equal to DELVT), it pauses after a form-feed character.

The "sg_kill" value defines line-cancel character for the character device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default line-cancel character is control-'x' (0x18).

The "sg_erase" value defines the character-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default character-cancel character is the backspace-character (control-'h', 0x08, ' h').

The value "sg_speed" is currently unused.

The bit-string "sg_prot" defines the type of start-stop protocol expected by the operating system for the character-device. The values defined in that bit-string are as follows:

ESC	0x80
OXON	0x40
ANY	0x20
TRANS	0x10
IXON	0x08

If ESC is set, the operating system stops writing to the character device when it reads an escape-character (0x1B) from the device. It resumes writing to the character-device when it reads another escape-character from the device. If OXON is set, the operating system stops writing to the character-device when it reads an xoff-character (0x13). It resumes writing to the character-device when it reads an xon-character (0x11). If ANY is set, the operating system uses any character read from the character-device as a substitute for the xon character. If TRANS is set, the operating system does not echo the escape-character, xoff-character, or xon-character to the the character-device. If IXON is set, the operating system writes a xoff character to the character-device whenever its internal buffers are full and it can't accept another character. It writes an xon-character to the character-device when space comes available for more characters.

The include-file "<sys/sgtty.h>" contains the structure and data definitions described above.

ERRORS REPORTED

EBADF The file descriptor is out of range or does not reference an open file

ENOTTY The file is not a character device

SEE ALSO

System Call: creat(), dup(), dup2(), open(), pipe(), stty()

Command: ttyset

kill

Send a signal to a task.

SYNOPSIS

```
#include <errno.h>
#include <sys/signal.h>
int kill(taskid, signum)
     int       taskid;
     int       signum;
```

Arguments

taskid The task-ID number of the task to receive the signal signum The signal to send the task

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function sends the signal numbered <signum> to the task whose task-ID number is <taskid>. A task may send a signal to another task only if its effective user is the system manager or it matches that of the specified task. The function returns zero if it successfully sent the task the specified signal, otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the signal number <signum> is out of range, there is no task with a task-ID number <taskid>, or the effective user of this task is not the system manager or does not match that of the specified task.

The include-file "<sys/signal.h>" defines the following constants and their meaning:

SIGHUP	1	Hang-up
SIGINT	2	Keyboard
SIGQUIT	3	Quit
SIGEMT	4	EMT OxA??? trap
SIGKILL	5	Task kill
SIGPIPE	6	Broken pipe
SIGSWAP	7	Swapping error
SIGTRACE	8	Trace
SIGALRM	10	Alarm
SIGTERM	11	Task terminate
SIGTRAPV	12	TRAPV instruction
SIGCHK	13	CHK instruction
SIGEMT2	14	EMT OxF??? emulation
SIGTRAP1	15	TRAP #1 instruction
SIGTRAP2	16	TRAP #2 instruction
SIGTRAP3	17	TRAP #3 instruction
SIGTRAP4	18	TRAP #4 instruction
SIGTRAP5	19	TRAP #5 instruction
SIGTRAP6	20	TRAP #6 instruction
SIGPAR	21	Parity error
SIGILL	22	Illegal instruction
SIGDIV	23	Division by zero
SIGPRIV	24	Privileged instruction
SIGADDR	25	Addressing error
SIGDEAD	26	A child task has died
SIGWRIT	27	Write to read-only memory
SIGEXEC	28	Execute from stack or data space
SIGBND	29	Segmentation violation
SIGUSR1	30	User defined signal #1
SIGUSR2	31	User defined signal #2
SIGUSR3	32	User defined signal #3
SIGABORT	33	Program abort
SIGSPLR	34	Spooler signal
SIGSYS3	35	System defined signal #3
SIGSYS4	36	System defined signal #4
SIGSYS5	37	System defined signal #5
SIGSYS6	38	System defined signal #6
SIGSYS7	39	System defined signal #7
SIGSYS8	40	System defined signal #8
SIGSYS9	41	System defined signal #9
SIGSYS10	42	System defined signal #10
SIGSYS11	43	System defined signal #11
SIGSYS12	44	System defined signal #12
SIGSYS13	45	System defined signal #13
SIGSYS14	46	System defined signal #14
SIGSYS15	47	System defined signal #15
SIGSYS16	48	System defined signal #16

SIGVEN1	49	Vendor defined signal #1
SIGVEN2	50	Vendor defined signal #2
SIGVEN3	51	Vendor defined signal #3
SIGVEN4	52	Vendor defined signal #4
SIGVEN5	53	Vendor defined signal #5
SIGVEN6	54	Vendor defined signal #6
SIGVEN7	55	Vendor defined signal #7
SIGVEN8	56	Vendor defined signal #8
SIGVEN9	57	Vendor defined signal #9
SIGVEN10	58	Vendor defined signal #10
SIGVEN11	59	Vendor defined signal #11
SIGVEN12	60	Vendor defined signal #12
SIGVEN13	61	Vendor defined signal #13
SIGVEN14	62	Vendor defined signal #14
SIGVEN15	63	Vendor defined signal #15

ERRORS REPORTED

EACCES The current effective user is not the system manager or the it does not match that of the specified task

EINVAL The signal number is out of range

ESRCH Invalid task number

SEE ALSO

System Call: signal()

Command: int

link

Create a link to a file.

SYNOPSIS

```
#include <errno.h>
int link(path, newlink)
     char     *path;
     char     *newlink;
```

Arguments

path	The address of a character-string containing a pathname for an existing file
newlink	The address of a character-string containing the pathname of the link to create

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function establishes a link to the file reached by the pathname in the character-string referenced by <path> called by the pathname in the character-string referenced by <newlink>. The pathname <path> must exist and the file it reaches may not be a directory unless the current effective user is the system manager. The pathname <newlink> must not exist. The device for <newlink> must be the same as that for <path>. The directory for <newlink> must give the current effective user writing permission. The function returns zero as its result if it successfully establishes the link. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if either the path in <path> or <newlink> can not be followed or contains a file which is not a directory, the pathname <path> does not exist, the pathname <newlink> already exists, the file reached by <path> is a directory and the current effective user is not the system manager, the directory for <newlink> does not grant the current effective user writing permission, or the link crosses devices.

ERRORS REPORTED

EACCES	The directory for <newlink> does not give the current effective user writing permission
EEXIST	The pathname <newlink> already exists
EISDR	The file reached by <path> is a directory and the current effective user is not the system manager
EMSDR	The path can not be followed for either <path> or <linkname>
ENOENT	The pathname <path> does not exist
ENOTDIR	The path in either <path> or <linkname> contains a file which is not a directory
EXDEV	Attempting to link across devices

NOTES

Linking to a file changes the last-access time of that file.

SEE ALSO

System Call: unlink()

Command: link, rename

lock

Lock a task in memory or unlock a locked task.

SYNOPSIS

```
#include <errno.h>
int lock(flag)
     int       flag;
```

Arguments

flag — A flag which indicates lock or unlock

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

If the value <flag> is not zero, this function locks the current task in memory, preventing the operating system from swapping the task to the system swap space. Otherwise, it unlocks the current task, permitting the operating system to swap the task to the system swap space if necessary. The current effective user must be the system manager.

The function returns zero if it succeeds, otherwise, it returns -1 with "errno" set to the system error code. The function fails if the current effective user is not the system manager.

ERRORS REPORTED

EACCES — The current effective user is not the system manager

NOTES

Unlocking a task that is not locked is not an error.

SEE ALSO

System Call: memman()

lrec

Add an entry to the operating system's lock table.

SYNOPSIS

```
#include <errno.h>
int lrec(fildes, count)
     int       fildes;
     int       count;
```

Arguments

fildes	The file descriptor for the file containing the record to lock
count	The number of bytes to lock from the current file position

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function adds an entry to the operating system's lock table for the open file referenced by the file descriptor <fildes>, locking a record beginning at the current file position containing <count> bytes. If the current task has an existing entry in the operating system's lock table for the same file descriptor, the function removes that entry. The function returns zero as its result if it successfully locks the record, otherwise it returns -1 with "errno" set to the system error code.

The function fails if the operating system's lock table contains an entry made by another task for the file referenced by <fildes> and the record locked by that entry contains all or part of the record this function is trying to lock or the operating system's lock table is full. It also fails if the file descriptor <fildes> is out of range, does not reference an open file, or references a file that is not a regular file.

ERRORS REPORTED

EBADF	The file descriptor is out of range, does not reference an open file, or references a pipe, character-special file (character-device), or a block-special file (blockdevice)
ELOCK	The record specified could not be locked because there already exists a lock on all or part of that record or the operating system's lock-table is full

NOTES

Locking a record only prevents others from locking it. This and other tasks may read or modify the record and may alter the file containing the record.

The function removes any existing lock table entry made by the current task for the specified file without regard to the eventual outcome of the function.

The function only permits one entry in the system lock table for each file a task has open.

The operating system removes all lock table entries made by a task when that task terminates.

SEE ALSO

System Call: creat(), dup(), dup2(), open(), pipe(), urec()

lseek

Change the current file position of an open file.

SYNOPSIS

```
#include <errno.h>
long lseek(fildes, offset, type)
     int       fildes;
     long      offset;
     int       type;
```

Arguments

fildes	The file descriptor of the file to reposition offset A count describing the offset of the new position type A value describing the offset type

Returns

The new offset from the beginning of the file if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the current file pointer in the file descriptor <fildes>, dependent upon the values <offset> and <type>. If <type> is 0, the function interprets <offset> as an absolute byte-count from the beginning of the file. If <type> is 1, it interprets <offset> as a byte-count relative to the current file position. If <type> is 2, it interprets <offset> as a byte-count relative to the end of the file. If the function successfully changes the current file pointer, it returns the new file position, which is the offset relative to the beginning of the file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor <fildes> is out of range, does not reference an open file, or references a file which can't be repositioned, such as a pipe or a character-special file. It also fails if the requested position is before the beginning of the file or the value <type> is not valid.

ERRORS REPORTED

EBADF	The file descriptor is out of range or does not reference an open file
EINVAL	The value <type> is not valid
ESEEK	The requested file position is before the beginning of the file or the file descriptor <fildes> references a file which can't be repositioned

NOTES

The function call "lseek(<fildes>,0,1)" returns as its result the file's current position.

The function does not change the file's current position if the function reports an error.

If the new position is beyond the current end of the file, the function creates a gap in the file which contains zeros if read. The function does not allocate any new blocks of media if the file resides on a block-device.

SEE ALSO

C Library: fseek()

System Call: creat(), dup(), dup2(), open(), pipe()

memman

Perform a memory management operation.

SYNOPSIS

```
#include <errno.h>
int memman(fcn, loaddr, hiaddr)
     int       fcn;
     char     *loaddr;
     char     *hiaddr;
```

Arguments

fcn	A value indicating the memory management function to perform
loaddr	The lowest address of memory to affect by the function
hiaddr	The highest address of memory to affect by the function

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function performs a memory management operation on the region of memory whose lowest address is <loaddr> and whose highest address is <hiaddr>. The value <fcn> selects which operation the function performs. The operations performed by this function are machine-dependent and may be different for the various implementation of the operating system. This function expects the current effective user to be the system manager. The function returns zero if it successfully performs the memory management function on the specified region of memory. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the function code <fcn> is out of range, if the high memory address <hiaddr> is lower than the low memory address <loaddr>, or if the current effective user is not the system manager. The function may also fail for reasons peculiar to the machine-dependent implementation of the function. The function has the following operations for the Tektronix 4044:

0 Clear the region's dirty-bit
1 Lock the region in memory
2 Unlock the region
3 Set write-protection on the region
4 Remove write-protection from the region
5 Release the memory allocated to the region

ERRORS REPORTED

EACCES The current effective user is not the system manager

EINVAL The function type is invalid or the starting address <loaddr> is higher than the ending address <hiaddr>

EVFORK The task shares its memory with its parent and may not call this function

mknod

Add an entry to the file-system that is a directory, a character-special file, or a block-special file.

SYNOPSIS

```
#include <errno.h>
int mknod(path, desc, devnum)
     char     *path;
     short     desc;
     short     devnum;
```

Arguments

path	The address of a character-string containing a pathname to the entry to create
desc	A bit-string describing the type of entry to create and the access permissions to assign to it
devnum	The major and minor device numbers to assign to the character-special or block-special file

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function adds an entry to the file-system that is a directory, a character-special file, or a block-special file. It gives the new entry the name found in the character-string referenced by <path>. The function determines the type of entry it creates from the bit-string <desc>. It assigns to that entry the access permissions described by the bit string <desc>, and if the entry is a character-special or block-special file, the function assigns to it the major and minor device numbers defined in the value <devnum>. The function ignores the argument <devnum> if it is creating a directory. The function requires that the current effective user be the system manager. The function returns zero if it successfully creates the entry in the file-system. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the pathname already exists, the path can't be followed, the path contains a file which is not a directory, the disk is full. It also fails if either the <desc> or <devnum> arguments are invalid, or the current effective user is not the system manager.

The bit-string <desc> describes the new entry's file type and its access permissions. It is defined as follows:

0x0001	Grant reading permission to the file's owner
0x0002	Grant writing permission to the file's owner
0x0004	Grant execution (or searching) permission to the file's owner
0x0008	Grant reading permission to other users
0x0010	Grant writing permission to other users
0x0020	Grant execution (or searching) permission to other users
0x0040	Set the task's effective user-ID to the owner-ID of the file if it is executed
0x0200	Make the file a directory
0x0400	Make the file a character-special file
0x0800	Make the file a block-special file

The function requires that the bit-string <desc> contain exactly one of the bit-values describing the file's type. It allows the bit-string to contain any of the bit-values defining the permissions, in any combination.

The argument <devnum> contains the major and minor device numbers to assign to the character-special or block-special file. The most significant byte contains the major device number, the least-significant byte contains the minor device number. This argument is ignored if the function creates a directory.

ERRORS REPORTED

EACCES	The current effective user is not the system manager
EEXIST	The pathname already references a file
EINVAL	The file description <desc> or the device number <devno> is not valid
EMSDR	The path to the file could not be followed
ENOSPC	The device is full
ENOTDIR	A part of the path is not a directory

NOTES

A character-special file is usually attached to a character-oriented device. Likewise, a block-special file is usually attached to a block-oriented device.

The third argument <devnum> should be specified as zero if <desc> indicates that the new entry is a directory.

SEE ALSO

System Call: creat()

Command: crdir, makdev

mount

Mount a block-special file onto the file-system.

SYNOPSIS

```
#include <errno.h>
int mount(spcnam, dirnam, rwflag)
     char     *spcnam;
     char     *dirnam;
     int       rwflag;
```

Arguments

spcnam	The address of a character-string containing a pathname to the block-special file to mount
dirnam	The address of a character-string containing a pathname to the directory to which to mount the blockspecial file
rwflag	A value indicating the type of accessing to permit

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function mounts the block-special file reached by the pathname in the character-string referenced by <spcnam> onto the directory reached by the pathname in the character-string referenced by <dirnam>. If the value <rwflag> is 0, the function mounts the file permitting reading and writing access. If <rwflag> is not 0, the function mounts the file so that it does not permit writing access. The function returns zero if it successfully mounts the specified block-special file onto the specified directory. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if it can't follow the path in <spcnam> or <dirnam>, the path in <spcnam> or <dirnam> contains a file which is not a directory, or either <spcnam> or <dirname> don't exist. It also fails if the file reached by <spcnam> isn't a block-special file, the file reached by <dirnam> isn't a directory, the block-special file reached by <spcnam> is already mounted, the directory reached by <dirnam> already has a block-special file mounted onto it, the operating system's mount table is full, or the current effective user is not the system manager. It also fails if the media associated with the block-special file was not unmounted correctly, or the media can not be read.

There is a block device associated with a block-special file. After mounting the block-special file, references to the directory onto which the file has been mounted now reference the root-directory of the media contained within that block device. If the file is being mounted permitting reading and writing access, the function sets an indicator on the media in the device associated with the file, indicating that the media is currently mounted. The "umount()" function clears this indicator.

ERRORS REPORTED

EACCES	The current effective user is not the system manager
EBUSY	The operating system's mount table is full or a device is already mounted on the specified directory <dirnam>
EDIRTY	The specified file <spcnam> was not properly unmounted and may be corrupt
EEXIST	The specified file <spcnam> is already mounted
EIO	The operating-system can not read the data on the device associated with the block-special file specified by <spcnam>
EMSDR	The path can not be followed for either <spcnam> or <dirnam>
ENOENT	There is no entry in the file-system for either <spcnam> or <dirnam>
ENOTDIR	The specified file <dirnam> is not a directory or the paths in either <spcnam> or <dirnam> contain a file that is not a directory

ENOTBLK The specified file <spcnam> is not a block-special file

NOTES

If this function reports EIO or EDIRTY errors, use "/etc/diskrepair" to try to salvage the data on the media that could not be mounted.

The function reports an EIO error if there is no media in the device associated with the block-special file, or if that media is not formatted correctly.

Disks written by the "backup" command can not be mounted.

SEE ALSO

System Call: mknod(), umount()

Command: /etc/backup, /etc/diskrepair, /etc/mount, /etc/unmount

nice

Change the task's scheduling priority.

SYNOPSIS

```
#include <errno.h>
int nice(incr)
     int  incr;
```

Arguments

incr The value to add to the task's priority

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the task's scheduling priority by adding the signed increment <incr> to it. The function permits the increment <incr> to be negative if the current effective user is the system manager. The function returns zero if it successfully changes the task's scheduling priority, otherwise -1 with "errno" set to the system error code.

The function fails if the increment <incr> is negative and the current effective user is not the system manager.

ERRORS REPORTED

EACCES	The increment <incr> is negative and the current effective user is not the system manager

NOTES

The function sets the task's scheduling priority to the maximum priority if the increment <incr> causes the priority to exceed the maximum. Likewise, the function sets the priority to the minimum priority if the increment causes the it to be less than the minimum.

open

Open an existing file.

SYNOPSIS

```
#include <errno.h>
#include <sys/fcntl.h>
int open(pathnam, mode)
     char     *pathnam;
     int       mode;
```

Arguments

pathnam	The address of a character-string containing a pathname to the file to open
mode	A value describing the requested access permissions

Returns

If successful, the file descriptor of the opened file, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function opens the file reached by the pathname in the character string referenced by <pathnam>, sets up access permissions described by the value <mode>, and sets the current file position to the beginning of the file. If the function succeeds, it returns as its result a file descriptor which references the open file. Other functions use this descriptor to reference the file opened by this function, such as "read()", "write()", "lrec()", and "fstat()", which manipulate files and their data. If the function fails, it returns -1 with "errno set to the system error code.

The function fails if the pathname can't be followed, the path contains a file which is not a directory, the pathname doesn't exist, the file doesn't grant the requested access permission to the current effective user, the task has the maximum number of files open, or the requested access permissions are invalid.

The value <mode> describes to the function the requested access permissions. If <mode> is O_RDONLY, the function opens the file for reading access. If <mode> is O_WRONLY, the function opens the file for writing access. If <mode> is O_RDWR, the function opens the file for both reading and writing access. The include-file "<sys/fcntl.h>" contains definitions for the constants O_RDONLY, O_WRONLY, and O_RDWR.

ERRORS REPORTED

EACCES	The file's permissions do not grant the requested access type
EINVAL	The value <mode> is invalid
EMFILE	The maximum number of files are open
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

A file descriptor is a non-negative integer that the operating system uses to reference an open file. It is an index into the operating system's open file table.

SEE ALSO

C Library: fclose(), fdopen(), fopen(), freopen()

System Call: close(), dup(), dup2(), fstat(), lrec(), pipe(), read(), write()

pause

Suspend the current task.

SYNOPSIS

```
include <errno.h>
int pause()
```

Arguments

None

Returns

This function always returns -1 with "errno" set to the error code EINTR

DESCRIPTION

This function suspends the current task indefinitely. This function returns only if the task receives a signal, catches that signal and returns from the function handling that signal, either explicitly using the return statement or implicitly by falling off the end of the function. It does not return if the task receives a signal that causes the task to terminate. Signals that the task ignores do not affect this function.

If the function returns it always returns -1 as its result with "errno" set to the system error code EINTR.

ERRORS REPORTED

EINTR	The task received a signal, causing it to resume execution

SEE ALSO

C Library: sleep()

System Call: alarm(), kill(), signal()

Command: sleep

phys

Access or release a system resource.

SYN

```
#include <errno.h>
char *phys(code)
     int       code;
```

Arguments

code A value identifying the resource

Returns

If successful and accessing a resource, it returns the logical address of the memory associated with the resource; if successful and releasing a resource, it returns (char *) -1; otherwise it returns (char *) NULL

DESCRIPTION

If the value <code> is greater than zero, this function accesses the resource identified by that value. If it successfully accesses the requested resource, it returns the logical address of the memory mapped for that resource as its result. Otherwise, it returns (char *) NULL as its result.

If the value <code> is less than zero, this function releases the resource identified by the absolute value of <code> and returns as its result (char *) -1.

If the value <code> is zero, this function releases all of the resources allocated by the current task through this function and returns as its result (char *) -1.

The resources that this function makes available depend on the particular implementation of the operating system, so the meaning of the value <code> differs from implementation to implementation. The following table describes the meaning of the absolute value of <code> for the Tektronix 4044:

1	The 128K bit-map
2	The first shared 4K page
3	The second shared 4K page
4	The time-of-day clock

ERRORS REPORTED

EINVAL The value <code> is out of range

NOTES

This function ignores requests to release resources that have not been allocated to the task. Likewise, it ignores requests to allocate resources that are already allocated by the task.

pipe

Create a pipe.

SYNOPSIS

```
#include <errno.h>
int pipe(fds)
     int     (*fds)[2];
```

Arguments

fds	The address of a two-element array of integers to receive the pipe's input output file descriptors

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function creates a pipe, which is a first-in, first-out I/O mechanism typically used to send data from one task to another. It saves the pipe's input file descriptor in the first element of the array referenced by <fds> and it saves the pipe's output file descriptor in the second element of that array. The function returns zero if it successfully creates a pipe, otherwise -1 with "errno" set to the system error code.

The function fails if the task has more than two less the permitted maximum number of open files.

Reading from a pipe whose buffers are not full and whose input file descriptor has not been closed suspends the task until the pipe is filled or the pipe's output file descriptor is closed. Writing to a pipe whose buffers are full suspends the task until all of the data written to the pipe has been read.

Reading from a pipe whose buffers contain no data and whose output file descriptor is closed causes the function attempting to read data from the pipe to report an end-of-file error. Writing to a pipe whose input file descriptor has been closed causes the function attempting to write the data to the pipe to report a broken pipe error.

Typically, a task creates a pipe using this function, then the task then executes a "fork()", duplicating the pipe's input and output file descriptors for the child (created) task. The sending task (the task that is to send data through the pipe) closes the pipe's input file descriptor and writes data to the pipe's output file descriptor. The receiving task (the task that is to receive data from the pipe) closes the pipe's output file descriptor and reads data from the pipe's input file descriptor.

ERRORS REPORTED

EMFILE The task has too many files open to create a pipe

NOTES

Undefined behavior results if a task attempts to use both the input file descriptor and the output file descriptor.

SEE ALSO

System Call: close(), fstat(), open(), read(), write()

Command: shell, script

profil

Start or stop monitoring the current task.

SYNOPSIS

```
int profil(bufad, bufsiz, scale, lowpc)
     char     *bufad;
     int       bufsiz;
     int       scale;
     int       lowpc;
```

Arguments

bufad	The address of the buffer to contain monitoring information
bufsiz	The number of bytes in the buffer whose address is <bufad>
scale	A value indicating the monitoring granularity
lowpc	The lowest address to monitor in the task

Returns

Zero

DESCRIPTION

If <scale> is not 0 or 1, this function requests that the operating system begin monitoring the current task, using the buffer whose address is <bufad> and contains <bufsiz> bytes, as the monitor buffer, beginning at the program address <lowpc>, with a granularity of <scale>. If <scale> is 0 or 1, this function requests that the operating system stop monitoring the current task. The function always returns zero as its result.

While monitoring a task, the operating system examines the task at each tick on the system clock, which occurs every tenth of a second. It takes the task's current program counter, subtracts from it the value <lowpc>, divides the result by <scale>, then multiplies the quotient by two. If the product is less than the value <bufsiz>, it adds the product to the address of the buffer <bufad>, then increments the word at that resulting address by one.

ERRORS REPORTED

None

NOTES

The buffer containing the values incremented at each clock tick must begin at an even address.

The operating system's monitoring mechanism only uses the least significant byte of the <scale> argument. After checking for 0 or 1, if <scale> is not a power of 2, it rounds the value up to the next power of 2 and uses that value as the scaling factor.

The argument <lowpc> is an address that has been cast into an int. The operating system automatically stops monitoring a task when that task calls the "exec()" function.

The operating system does not automatically stop monitoring a task when that task calls the "fork()" function.

SEE ALSO

C Library: monitor()

System Call: exec(), fork()

Command: cc

read

Read data from an open file.

SYNOPSIS

```
#include <errno.h>
int read(fildes, bufad, nbytes)
int       fildes;
char     *bufad;
int       nbytes;
```

Arguments

fildes	A file descriptor for the open file from which to read data
bufad	The address of a buffer to contain the data read
nbytes	The maximum number of byte to read

Returns

The number of bytes read if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function reads data from the open file referenced by the file descriptor <fildes>, beginning at the current file position, reading a maximum of <nbytes> bytes, writing the data read into the buffer whose address is <bufad>. The function reads data until it reads the maximum number of bytes, it reaches the end of the associated file, or, if the associated file is a character-special file (terminal), the function reads an end-of-line character. If the associated file is one that can be repositioned, the function changes the current file position to that of the data immediately following the last byte read.

If the function successfully reads data, it returns the number of bytes it read as its result. If the function encounters the end of the file before reading any data, it returns zero as its result. Otherwise, it returns -1 as its result and sets "errno" to the system error code describing the error.

The function fails if the file descriptor <fildes> is out of range, does not reference an open file, or references an open file that is not open for reading. It also fails if an I/O error occurs while reading data or the requested count <nbytes> is negative. The function also fails if the task receives and catches a signal while reading data from a slow device, such as a terminal.

ERRORS REPORTED

EACCES	The specified file is not opened for reading
EBADF	The file descriptor is out of range or does not reference an open file
EBARG	The value <nbytes> is not valid
EINTR	The task received and caught a signal while the function was reading from a slow device
EIO	The operating system reports an I/O error

NOTES

The data in the buffer may change if the function reports an I/O error.

SEE ALSO

C Library: fread()

System Call: creat(), dup(), dup2(), open(), pipe(), write()

sbrk

Change the data segment's memory allocation.

SYNOPSIS

```
#include <errno.h>
char *sbrk(incr)
int       incr;
```

Arguments

incr	The number of bytes to enlarge or shrink the data segment

Returns

The data segment's end-of-segment address before it was enlarged or shrunk if successful, otherwise (char *) -1 with "errno" set to the system error code

DESCRIPTION

This function enlarges or shrinks the memory allocation of the current task's data segment by <incr> bytes. If <incr> is positive, it enlarges the segment. If <incr> is negative, it shrinks the segment. If <incr> is zero, it does not change the segment's memory allocation. If the function succeeds, it returns as its result the end-of-segment address for the data segment before the function changed the segment's memory allocation. If <incr> is positive, this is the address of the first byte of newly allocated memory. If <incr> is negative, this address meaningless since it references memory that is out of the task's address space. If <incr> is zero, this address is the current end-of-segment address of the data segment. Otherwise, it returns (char *) -1 with "errno" set to the system error code indicating the error.

The function fails if it couldn't allocate enough memory to make the data segment larger by <incr> bytes. It also fails if <incr> is negative and the absolute value of <incr> is larger than the number of bytes allocated to the data segment.

ERRORS REPORTED

ENOMEM	The function can't allocate enough memory to enlarge the data segment by <incr> bytes, or there isn't enough memory allocated to the segment to shrink it by the requested number of bytes

NOTES

The function does not change the data segment's memory allocation if it reports an error.

The end-of-segment address is the lowest address that is higher than the highest address of memory allocated to the segment.

The function returns the current end-of-segment address without changing the segment's memory allocation if <incr> is zero.

SEE ALSO

C Library: calloc(), EDATA, free(), malloc(), realloc()

System Call: brk(), cdata()

set_ftm

Change a file's last-modification time.

SYNOPSIS

```
#include <errno.h>
int set_ftm(pathnam, ptime)
char     *pathnam;
long     *ptime;
```

Arguments

pathnam	The address of a character-string containing a pathname for the file whose modification time is to change
ptime	The address of the value to set as the file's modification time

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the last-modification time for the file reached by the pathname in in the character-string referenced by <pathnam> to the system-time value referenced by <ptime>. The function requires that the current effective user be the system manager. The function returns zero if it successfully changes the file's modification time, otherwise -1 with "errno" set to the system error code.

This function fails if the path in <pathnam> can't be followed or contains a file which isn't a directory. It also fails if the pathname doesn't exist, the file reached by the pathname is currently open, or the current effective user isn't the system manager.

ERRORS REPORTED

EACCES	The current effective user is not the system manager
EBADF	The file descriptor is out of range or does not reference an open file
EBUSY	The specified file is currently open
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

This function is obsolete and is included only for compatibility. New applications should use "utime()".

The operating system measures time as a count of seconds since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

This function does not compare the new modification time with the creation time or the current time, so it is possible to set the file's modification time to before it's creation date or to some time in the future.

SEE ALSO

System Call: fstat(), stat(), utime()

Command: ls

setuid

Change both the user-ID and the effective user-ID.

SYNOPSIS

```
#include <errno.h>
int setuid(uid)
int       uid;
```

Arguments

uid	The user-ID of the new user and effective user

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the task's current user-ID and the task's current effective user-ID to <uid>. The function expects either the current user or the current effective user to be the system manager. The function returns zero as its result if it successfully changes the task's user-ID and effective user-ID. Otherwise, it returns -1 with "errno" set to the system error code.

This function fails if neither the current user nor the current effective user is the system manager.

ERRORS REPORTED

EACCES	Neither the current user nor the current effective user is the system manager

SEE ALSO

System Call: geteuid(), getuid()

Command: login, who

signal

Change the signal-handling address for a specific signal in the current task.

SYNOPSIS

```
#include <errno.h>
#include <sys/signal.h>
int (*signal(signum, handler))()
int       signum;
int     (*handler)();
```

Arguments

signum	The signal number for which signal handling is being changed
handler	The new signal-handling address for the specified signal

Returns

The previous signal-handling address for the specified signal if successful, otherwise (int (*)()) -1 with "errno" set to the system error code

DESCRIPTION

This function changes the signal-handling address for the specified signal <signum> in the current task to the function whose address is <handler>. If it succeeds, it returns as its result the previous signal-handling address for the specified function. If it fails, it returns as its result (int (*)()) -1 with "errno" set to the system error code.

The value SIG_IGN is a special signal-handling address which, if passed to this function as the signal-handling address <handler>, tells the function that the task is to ignore the specified signal. If the function returns this value, the task was ignoring the specified signal. The value SIG_DFL is a special signal-handling address which, if passed to this function as the signal-handling address <handler>, tells the function that the task is to terminate if it receives the specified signal. If the function returns this value, the task would have terminated if it received the specified signal. The include-file "<sys/signal.h>" contains the definitions for SIG_IGN and SIG_DFL.

That include-file also defines constants for each of the sixty-three signals defined by the operating system. These constants and the signal associated with the constants are as follows:

SIGHUP	1	Hang-up
SIGINT	2	Keyboard
SIGQUIT	3	Quit
SIGEMT	4	EMT OxA??? trap
SIGKILL	5	Task kill
SIGPIPE	6	Broken pipe
SIGSWAP	7	Swapping error
SIGTRACE	8	Trace
SIGALRM	10	Alarm
SIGTERM	11	Task terminate
SIGTRAPV	12	TRAPV instruction
SIGCHK	13	CHK instruction
SIGEMT2	14	EMT OxF??? emulation
SIGTRAP1	15	TRAP #1 instruction
SIGTRAP2	16	TRAP #2 instruction
SIGTRAP3	17	TRAP #3 instruction
SIGTRAP4	18	TRAP #4 instruction
SIGTRAP5	19	TRAP #5 instruction
SIGTRAP6	20	TRAP #6 instruction
SIGPAR	21	Parity error
SIGILL	22	Illegal instruction
SIGDIV	23	Division by zero
SIGPRIV	24	Privileged instruction
SIGADDR	25	addressing error
SIGDEAD	26	A child task has died
SIGWRIT	27	Write to read-only memory
SIGEXEC	28	Execute from stack or data space
SIGBND	29	Segmentation violation
SIGUSR1	30	User defined signal #1
SIGUSR2	31	User defined signal #2
SIGUSR3	32	User defined signal #3
SIGABORT	33	Program abort
SIGSPLR	34	Spooler signal
SIGSYS3	35	System defined signal #3
SIGSYS4	36	System defined signal #4
SIGSYS5	37	System defined signal #5
SIGSYS6	38	System defined signal #6
SIGSYS7	39	System defined signal #7
SIGSYS8	40	System defined signal #8
SIGSYS9	41	System defined signal #9
SIGSYS10	42	System defined signal #10
SIGSYS11	43	System defined signal #11
SIGSYS12	44	System defined signal #12
SIGSYS13	45	System defined signal #13
SIGSYS14	46	System defined signal #14

SIGSYS15	47	System defined signal #15
SIGSYS16	48	System defined signal #16
SIGVEN1	49	Vendor defined signal #1
SIGVEN2	50	Vendor defined signal #2
SIGVEN3	51	Vendor defined signal #3
SIGVEN4	52	Vendor defined signal #4
SIGVEN5	53	Vendor defined signal #5
SIGVEN6	54	Vendor defined signal #6
SIGVEN7	55	Vendor defined signal #7
SIGVEN8	56	Vendor defined signal #8
SIGVEN9	57	Vendor defined signal #9
SIGVEN10	58	Vendor defined signal #10
SIGVEN11	59	Vendor defined signal #11
SIGVEN12	60	Vendor defined signal #12
SIGVEN13	61	Vendor defined signal #13
SIGVEN14	62	Vendor defined signal #14
SIGVEN15	63	Vendor defined signal #15

ERRORS REPORTED

EINVAL The value <signum> is not a valid signal number

NOTES

The function "signal()" is a function returning a pointer to a function returning an int.

The argument <handler> is a pointer to a function returning an int.

The function does not verify the argument <handler> to ensure that no memory-violation or bus-error occurs if the specified signal is caught.

The operating system produces a core image in a file called "core" in the working directory if the default action (termination) is taken by a task on receipt of certain signals and other conditions are met. Those certain signals are: SIGABORT, SIGADDR, SIGBND, SIGCHK, SIGEMT, SIGEMT2, SIGDIV, SIGEXEC, SIGILL, SIGQUIT, SIGPAR, SIGPRIV, SIGSWAP, SIGTRAP1, SIGTRAP2, SIGTRAP3, SIGTRAP4, SIGTRAP5, SIGTRAP6, SIGTRAPV, SIGWRIT. The operating system generates a core image file only if user-ID and the actual user-ID of the task receiving the signal are the same and there there is an existing file in the working directory called "core" which gives the current effective user writing permission or the working directory gives the current effective user writing permission. The operating system does not permit tasks to catch or ignore some signals. Those signals are: SIGABORT, SIGKILL.

SEE ALSO

System Call: kill()

Command: int

stack

Check and expand memory allocated to the task's stack segment.

SYNOPSIS

```
#include <errno.h>
int stack(nbytes)
int       nbytes;
```

Arguments

nbytes	The number of bytes that the stack is expected to grow

Returns

Zero if the stack segment has enough space to contain a program stack enlarged by <nbytes> bytes, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function guarantees that the the task has enough memory allocated to its stack segment so that the program stack can expand by <nbytes> bytes. If the segment is not large enough, the function allocates enough memory to the segment so that the stack can expand by the specified amount. The function returns zero if the stack segment has enough space allocated to it to contain a program stack enlarged by <nbytes> bytes. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if can't allocate more memory to the task's stack segment.

ERRORS REPORTED

ESTOF	The task's stack segment is as large as it can get
EVFORK	The task shares its memory with its parent and may not call this function

NOTES

Unless otherwise directed, the C-compiler automatically generates code which ensures stack integrity.

SEE ALSO

Command: cc

stat

Get the status of a file.

SYNOPSIS

```
#include <errno.h>
#include <sys/stat.h>
int stat(pathnam, bufad)
char          *pathnam;
struct stat   *bufad;
```

Arguments

pathnam	The address of a character-string containing a pathname for the file to examine
bufad	The address of the structure to contain the file's status

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function examines the file reached by the pathname in the character-string referenced by <pathnam> and writes information describing the status of that file into the structure whose address is <bufad>. The function returns zero as its result if it successfully gets the status of the file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the path in <pathnam> can't be followed or if it contains a file which isn't a directory. It also fails if the pathname doesn't reach a file.

The following structure is defined in the include-file "<sys/stat.h>" and defines the format of the data describing the status of the file:

```
struct stat

{
short     st_dev;
short     st_ino;
char      st_filler;
char      st_mode;
char      st_perm;
char      st_nlink;
short     st_uid;
long      st_size;
long      st_mtime;
long      st_spr;
};
```

The value "st_dev" is the device number of the device containing the file; "st_ino" is the file descriptor number (FDN) on the device describing the file; "st_filler" is an unused byte; "st_mode" is a bit string describing the type of the file, described below; "st_perm" is a bit-string describing the permissions of the file, described below; "st_nlink" is the number of links to the file; "st_uid" is the owner-ID of the file; "st_size" is the size of the file, in bytes; "st_mtime" is the last modification date and time for the file, in system-time; and "st_spr" is unused.

The following constants, defined in the include-files "<sys/stat.h>" and "<sys/modes.h>", define the data in the bit-string "st_mode" which describes the file's type:

```
S_IFMT    0x4F
S_IFREG   0x01
S_IFBLK   0x03
S_IFCHR   0x05
S_IFDIR   0x09
S_IFPIPE  0x41
```

The constant S_IFMT is a mask that when anded with the value "st_mode" yields the file type. After anding with the constant S_IFMT "st_mode" produces S_IFREG if the file is a regular file, S_IFBLK if the file is a block-special file (block device), S_IFCHR if the file is a character-special file (character device), S_IFDIR if the file is a directory, or S_IFPIPE if the file is a pipe.

The following constants, also defined in the include-files "<sys/stat.h>" and "<sys/modes.h>", define the data in the bit-string "st_perms" which describes the file's access permissions:

```
S_IREAD   0x01
S_IWRITE  0x02
S_IEXEC   0x04
S_IOREAD  0x08
S_IOWRITE 0x10
S_IOEXEC  0x20
S_ISUID   0x40
```

The value S_IREAD grants reading permission to the owner of the file, S_WRITE grants writing permission to the owner, and S_IEXEC grants searching permission to the owner if the file is a directory, otherwise it grants execution permission. The value S_IOREAD grants reading permission to users other than the owner of the file, S_IOWRITE grants writing permission to others, and S_EXEC grants searching permission to others if the file is a directory, otherwise it grants execution permission. The value S_ISUID causes the effective user-ID to be changed to that of the owner of the file whenever the program contained in the file is executed.

ERRORS REPORTED

EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

The include-file "<sys/modes.h>" need not be included if the includefile "<sys/stat.h>" since "<sys/stat.h>" includes "<sys/modes.h>".

SEE ALSO

System Call: creat(), dup(), dup2(), fstat(), open(), pipe(), utime()

Command: ls

stime

Set the system-time value.

SYNOPSIS

```
#include <errno.h>
int stime(ptime)
long     *ptime;
```

Arguments

ptime	The address of the value to set as the new system-time value

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the operating system's system-time value, its time-of-day value, to the value referenced by <ptime>. The function requires that the current effective user be the system manager. The function returns zero as its result if it successfully sets the system-time value. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the current effective user is not the system manager.

ERRORS REPORTED

EACCES	The current effective user is not the system manager

NOTES

The operating system represents the time of day as the number of seconds that has elapsed since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

SEE ALSO

System Call: time(), times()

Command: date

stty

Set the characteristics of an open character-device.

SYNOPSIS

```
#include <errno.h>
#include <sys/sgtty.h>
int stty(fildes, buf)
int             fildes;
struct sgttyb *buf;
```

Arguments

fildes	A file descriptor for the open character-device
buf	The address of the structure to contain the new characteristics

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the characteristics of the open character-device referenced by the file descriptor <fildes> and to those described by the data in the structure referenced by <buf>. The function returns zero as its result if it successfully changes the characteristics of the open character-device. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor is out of range, does not reference an open file, or does not reference an open character-device.

The function call expects <buf> to be the address of a structure which is defined as follows:

```
struct sgttyb
{
unsigned char  sg_flag;
unsigned char  sg_delay;
unsigned char  sg_kill;
unsigned char  sg_erase;
unsigned char  sg_speed;
unsigned char  sg_prot;
};
```

The bit-string "sg_flag" describes the current mode of the terminal. The values in the bit-string are as follows:

RAW	0x01
ECHO	0x02
XTABS	0x04
LCASE	0x08
CRMOD	0x10
SCOPE	0x20
CBREAK	0x40
CNTRL	0x80

If RAW is set, the operating system considers the character-device to be in raw mode. In raw mode, the operating system suspends all processing of input and output. If clear, the operating system considers the character-device to be in non-raw mode (sometimes called "cooked mode"). In this mode, the operating system processes characters dependent upon the setting of the other bits in the bit-string.

If ECHO is set, the operating system echoes characters read to the character-device. If clear, the operating system doesn't echo characters to the device. If XTABS is set, the operating system expands tab characters to spaces during output operations so that the next character written to the device is written to a column number that is an even multiple of eight. If clear, the operating system writes tab-characters to the character-device with no expansion. Tab-characters are defined by the system to be 0x09 and are defined by the C compiler as ' t'.

If LCASE is set, the operating system changes all upper-case characters to lower-case characters during input operations and changes all lowercase characters to upper-case characters during output operations. If clear, the operating system disables this feature. If CRMOD is set, the operating system writes a line-feed character to the character-device after every carriage-return character written. If clear, the operating system disables this feature.

If SCOPE is set, the operating system writes a backspace-character (0x08) followed by a space-character followed by another backspace character to the character-device whenever a character-cancel character is read from the character-device. If clear, the operating system disables this feature. If CBREAK is set, the operating system considers the terminal to be in single-character mode. In this mode, the operating system reads data from the device one character at a time, passing each character to the calling task. If clear, the operating system considers the terminal to be in line mode, where it reads data from the device one line at a time, passing data to the calling task whenever a terminator is read.

If CNTRL is set, the operating system ignores all characters read from the character-device that are outside of the range 0x20 through 0x7E inclusive, except for the line-terminator character (carriage return), the keyboard-interrupt character (control-'c'), the quit-interrupt character (control-' '), the character-cancel character, the line-cancel character, and the output-stop and output-start characters if any.

The bit-mask "sg_delay" indicates which characters, if written to the character-device, causes the operating system to pause before writing another character to the character-device. The values in that bit string are as follows:

DELNL	0x03
DELCR	0x0C
DELTB	0x10
DELVT	0x20
DELFF	0x20

If DELNL is set, the operating system pauses it writes a new-line character to the character-device. If DELCR is set, it pauses after a carriage-return character, if DELTB is set, it pauses after a horizontal-tab character, if DELFF is set (which is equal to DELVT), it pauses after a form-feed character.

The "sg_kill" value defines line-cancel character for the character device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default line-cancel character is control-'x' (0x18).

The "sg_erase" value defines the character-cancel character for the character-device. The operating system treats this character like any other character if the character-device is in single-character or raw mode. The default character-cancel character is the backspace-character (control-'h', 0x08, ' h').

The value "sg_speed" is currently unused.

The bit-string "sg_prot" defines the type of start-stop protocol expected by the operating system for the character-device. The values defined in that bit-string are as follows:

ESC	0x80
OXON	0x40
ANY	0x20
TRANS	0x10
IXON	0x08

If ESC is set, the operating system stops writing to the character device when it reads an escape-character (0x1B) from the device. It resumes writing to the character-device when it reads another escape-character from the device. If OXON is set, the operating system stops writing to the character-device when it reads an xoff-character (0x13). It resumes writing to the character-device when it reads an xon-character (0x11). If ANY is set, the operating system uses any character read from the character-device as a substitute for the xon character. If TRANS is set, the operating system does not echo the escape-character, xoff-character, or xon-character to the the character-device. If IXON is set, the operating system writes a xoff character to the character-device whenever its internal buffers are full and it can't accept another character. It writes an xon-character to the character-device when space comes available for more characters.

The include-file "<sys/sgtty.h>" contains the structure and data definitions described above.

ERRORS REPORTED

EBADF — The file descriptor is out of range or does not reference an open file

ENOTTY — The file is not a character device

SEE ALSO

System Call: creat(), dup(), dup2(), gtty(), open(), pipe()

Command: ttyset

sync

Update the file-system.

SYNOPSIS

```
int sync()
```

Arguments

None

Returns

Zero

DESCRIPTION

This function updates the file-system so that the media match the internal description of the file-system. The function always returns zero as its result.

ERRORS REPORTED

None

SEE ALSO

Command: update

time

Get the current system-time value.

SYNOPSIS

```
long time(ptime)
long     *ptime;
```

Arguments

ptime	The address of the long to receive the system-time value or (long *) NULL

Returns

The current system-time value.

DESCRIPTION

This function gets the current system-time value, the current time- of-day value, from the operating system and returns that value as its result. If <ptime> is not (long *) NULL, the function stores the system-time value at the location referenced by <ptime>.

ERRORS REPORTED

None

NOTES

The operating system represents time as the number of seconds that has elapsed since the epoch. It defines the epoch as 00:00 (midnight) January 1, 1980 Greenwich Mean Time.

SEE ALSO

System Call: stime(), times()

Command: date

times

Get the current task's CPU-usage information.

SYNOPSIS

```
#include <sys/times.h>
int times(ptimes)
struct tms    *ptimes;
```

Arguments

ptimes	The address of the structure to receive the task's current CPU-usage information

Returns

Zero

DESCRIPTION

This function gets the current task's CPU-usage information and places that information in the structure whose address is <ptimes>. The Cpu usage information includes measurements of the task's Central Processing Unit (CPU) use, the operating system's CPU use on behalf of the task, the task's children's total CPU use, and total operating system's Cpu use on behalf of the task's children. The system measures CPU use in hundredths of a second. The function always returns zero as its result.

The function expects <ptimes> to be the address of a structure which is defined as follows:

```
struct tms
{
long        tms_utime;
long        tms_stime;
long        tms_cutime;
long        tms_cstime;
};
```

The value "tms_utime" contains CPU-time used by the current task, "tms_stime" contains the CPU-time used by the operating system in behalf of the current task, "tms_cutime" contains the total CPU-time used by all of the task's descendants which have terminated, and "tms_cstime" contains the total CPU-time used by the operating system in behalf of all of the task's descendants which have terminated. The include-file "<sys/times.h>" defines this structure.

ERRORS REPORTED

None

NOTES

The operating system updates the task's CPU-usage information for its descendants whenever a direct descendants task terminates. The operating system continuously updates the task's CPU-usage information for itself.

SEE ALSO

System Call: fork(), time(), vfork()

Command: script, shell

truncf

Set the size of an open file.

SYNOPSIS

```
#include <errno.h>
int truncf(fildes)
int       fildes;
```

Arguments

fildes	A file descriptor for the file whose size is to change

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function sets the size of the open file referenced by the file descriptor <fildes> so that its end-of-file is the current file position. If that position is before the existing end-of-file, the function truncates the file. If that position is beyond the existing end- of-file, the function extends the file. The function returns zero as its result if it successfully sets the size of the specified file. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor <fildes> is not a valid file descriptor, does not reference an open file, or does not reference a file that has been opened for writing.

ERRORS REPORTED

EACCES	The file descriptor references a file that is not open for writing
EBADF	The file descriptor is out of range or does not reference an open file

NOTES

If the function truncates the file, all data beyond the new end-of-file is lost.

If the function extends the file, it does so without allocating to the file any blocks of the medium on which the file resides. Functions reading from the extended space read zeros.

SEE ALSO

System Call: creat(), dup(), dup2(), open(), pipe()

ttyslot

Get the terminal number of the task's controlling terminal.

SYNOPSIS

```
int ttyslot()
```

Arguments

None

Returns

The terminal number of the task's controlling terminal or zero if none

DESCRIPTION

The function gets the terminal number of the task's controlling terminal and returns that value as its result. If the task has no controlling terminal, the function returns zero as its result.

ERRORS REPORTED

None

NOTES

The operating system detaches a task from its controlling terminal if the task closes its standard input file, its standard output file, and its standard error file.

SEE ALSO

C Library: ttyname()

umask

Change the task's file-creation permissions mask.

SYNOPSIS

```
int umask(perms)
int       perms;
```

Arguments

perms	A bit-string containing the new file-creation permissions mask

Returns

The previous value of the file-creation permissions mask

DESCRIPTION

This function changes the task's file-creation permissions mask to the low-order six bits of the bit-string <perms>. It returns as its result the previous value of the task's file-creation permissions mask.

The file-creation permissions mask describes the permissions that may not be applied to a created file. The file-creation function, "creat()", ands the one's-complement of the task's file-creation permissions mask with the bit-string describing the permissions for the file being created, and applies the resulting permissions bit-string to the created file.

ERRORS REPORTED

None

NOTES

A task inherits its file-creation permissions mask from its parent.

SEE ALSO

System Call: creat(), fork(), fstat(), stat(), vfork()

umount

Unmount a mounted device.

SYNOPSIS

```
#include <errno.h>
int umount(pathnam)
char     *pathnam;
```

Arguments pathnam A character-string containing a pathname to the device to unmount

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function unmounts the mounted device reached by the pathname in the character-string referenced by <pathnam>. The function returns zero if it successfully unmounts the device, Otherwise, it returns -1 with "errno" set to the system error code.

This function fails if the path in the pathname <pathnam> can't be followed or contains a file which isn't a directory. It also fails the pathname doesn't reach a file, the file it reaches isn't a device, the device is not mounted, or the device is busy.

ERRORS REPORTED

EBDEV	The pathname <pathnam> reaches something other than a device
EBUSY	The device is busy
EMSDR	The path in the pathname <pathnam> can't be followed
ENMNT	The specified device is not mounted
ENOENT	The pathname <pathnam> does not reach a device

NOTES

A device that was mounted for read and write access but was not unmounted correctly can't be mounted again until it is repaired by "/etc/diskrepair".

A device-busy error (EBUSY) usually indicates that a file on the specified device is currently open or that a task has as its working directory a directory on the device.

SEE ALSO

System Call: mount()

Command: /etc/diskrepair, /etc/mount, /etc/unmount

unlink

Remove a link to a file.

SYNOPSIS

```
#include <errno.h>
int unlink(pathnam)
char     *pathnam;
```

Arguments

pathnam	The address of a character-string containing the pathname of the link to be removed

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function removes the pathname in the character-string referenced by <pathnam>. If that pathname is the last that reaches the associated file, the operating system deletes the file. The function returns zero as its result if it successfully removes the pathname. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the path in the pathname <pathnam> can't be followed or it contains a file which isn't a directory. It also fails if the pathname doesn't exist the directory containing the pathname doesn't grant the current effective user writing permission.

ERRORS REPORTED

EACCES	The directory containing the specified file doesn't grant writing permission to the current effective user
EMSDR	The path to the file could not be followed
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

This function can remove any entry in any directory which grants writing permission to the current effective user. That entry can be a directory which is not empty and can be the directories "." and "..".

If this function removes the pathname to an open file, the operating system postponed deleting that file until it is closed.

SEE ALSO

System Call: creat(), link()

Command: create, kill, link

urec

Remove an entry from the operating system's lock table.

SYNOPSIS

```
#include <errno.h>
int urec(fildes)
int       fildes;
```

Arguments

fildes	The file descriptor whose lock table entry the function is to remove

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function removes from the operating system's lock table the task's entry for the file referenced by the file descriptor <fildes>. The function returns zero if it successfully removes the entry, otherwise -1 with "errno" set to the system error code.

The function fails if the file descriptor <fildes> is not a valid file descriptor or does not reference an open file.

ERRORS REPORTED

EBADF	The file descriptor is out of range or does not reference an open file

NOTES

The function returns zero as its result if there is no entry in the operating system's lock table for the specified file descriptor.

The operating system permits only one lock per file descriptor for each task.

Placing a lock on a portion of a file stops other tasks from placing a lock on the same portion of that file. It does not stop reading from and writing to that portion of the file.

SEE ALSO

System Call: lrec()

utime

Change a file's last-modification time.

SYNOPSIS

```
#include <errno.h>
int utime(pathnam, ptime)
char     *pathnam;
long     *ptime;
```

Arguments

pathnam	The address of a character-string containing a pathname for the file whose modification time is to change
ptime	The address of the the value to set as the file's modification time

Returns

Zero if successful, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function changes the last-modification time for the file reached by the pathname in in the character-string referenced by <pathnam> to the system-time value referenced by <ptime>. The function requires that the current effective user be the system manager. The function returns zero if it successfully changes the file's modification time, otherwise -1 with "errno" set to the system error code.

This function fails if the path in <pathnam> can't be followed or contains a file which isn't a directory. It also fails if the pathname doesn't exist, the file reached by the pathname is currently open, or the current effective user isn't the system manager.

ERRORS REPORTED

EACCES	The current effective user is not the system manager
EBADF	The file descriptor is out of range or does not reference an open file
EBUSY	The specified file is currently open
ENOENT	The pathname does not reach a file
ENOTDIR	A part of the path is not a directory

NOTES

The operating system measures time as a count of seconds since the epoch. It defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time.

This function does not compare the new modification time with the creation time or the current time, so it is possible to set the file's modification time to before it's creation date or to some time in the future.

SEE ALSO

System Call: fstat(), stat()

Command: ls

vfork

Create a new task.

SYNOPSIS

```
#include <errno.h>
int vfork()
```

Arguments

None

Returns

If successful, the child's task-ID to the parent (calling) task and zero to the child (created) task, otherwise -1 with "errno" set to the system error code

DESCRIPTION

This function creates a new task (the child task) that is an exact copy of the current task (the parent task). If the function succeeds, it returns the task-ID of the child to the parent task and returns zero to the child task. Otherwise, it returns -1 with "errno" set to the system error code.

This function fails if the current user can't allocate another task, the system task table is full, or if the current task may not call this function.

This function differs from the "fork()" function call in that it generates the new task more efficiently on a virtual memory system. Instead of making a copy of the data in the parents memory, the child task inherits the memory allocated to the parent task. The child task isn't allowed to call the "fork()", "memman()", or "vfork()" functions or any function that may change the memory configuration such as "sbrk()" or "stack()" until after it executes a program using the "execl()", " execlp()", "execv()", or "execvp()" functions. The function doesn't return to the parent task until the child task terminates or executes a program.

The child task is identical to the parent task in that it has the same task priority, user-ID, effective user-ID, controlling terminal information, file-creation permissions mask, working directory, signal handling set-up, profiling information, and allocated memory.

The child task differs from the parent task in that its task-ID is different, its parent task-ID is the task-ID of the parent task, its file descriptors are exact copies of those in the parent task, and its system and user CPU times are reset to zero.

ERRORS REPORTED

EAGAIN	The maximum number of tasks for the user are active or there are no available entries in the system task table
EVFORK	The task shares its memory with its parent and may not call this function

NOTES

A task-ID is a non-negative integer.

This function is the same function as "fork()" on systems which aren't virtual memory systems.

The child task shares its stack with its parent so it shouldn't return from the scope which calls the "vfork()" function.

SEE ALSO

C Library: exit(), _exit()

System Call: execl(), execlp(), execv(), execvp(), fork(), memman(), wait()

wait

Suspend the task until a child task terminates.

SYNOPSIS

```
#include <errno.h>
int wait(ptaskid)
int      *ptaskid;
```

Arguments

ptaskid	The address of the int to get the termination status of the child task that terminated

Returns

The task-ID of the terminated task or -1 with "errno" set to the system error code

DESCRIPTION

This function suspends the current task until a child task terminates. When a child task terminates, the function puts the termination status of the terminated child into the value referenced by <ptaskid> and returns as its result the task-ID of the terminated child task.

If the function returns -1, it didn't wait for a child task to terminate. The function returns without a child task terminating if there are no active child tasks or the task catches a signal.

The termination status contains the child task's exit code, the signal number that caused its termination, and a flag that indicates that it produced a core image file (a core dump). Anding the termination status with 0x00FF extracts from it the child task's exit code. This is the low-order 8-bits of the argument to "exit()" (or "_exit()") which terminated the task and typically indicates an error if it is not zero. Anding the termination status with 0x7F00 extracts from it the signal number that caused its termination. This is only non-zero if the child task did not terminate using the "exit()" or "_exit()" functions. Anding the termination status with 0x8000 extracts from it the core-image flag. If the flag is not zero, the child task produced a core image file when it terminated. Otherwise, it did not produce a core image file.

ERRORS REPORTED

ECHILD	There are no child tasks active
EINTR	The task caught a signal and that caused this function to end abnormally

SEE ALSO

C Library: sleep()

System Call: alarm(), fork(), kill(), pause(), signal()

Command: int, wait

write

Write data to an open file.

SYNOPSIS

```
#include <errno.h>
int write(fildes, bufad, nbytes)
int       fildes;
char     *bufad;
int       nbytes;
```

Arguments

fildes	A file descriptor for the open file to which the data is to be written
bufad	The address of the buffer containing the data to write
nbytes	The number of bytes of data to write

Returns

The number of bytes of data written to the file or -1 if none with "errno" set to the system error code

DESCRIPTION

This function writes data to the file referenced by the file descriptor <fildes>. It writes data to the file from the buffer whose address is <bufad> <nbytes> bytes of data. The function successfully writes the data, it returns as its result the number of bytes of data that it wrote. Otherwise, it returns -1 with "errno" set to the system error code.

The function fails if the file descriptor is out of range, references a file that isn't open for writing, or references a broken pipe. The function also fails if the disk is full or the operating system reports an I/O error while writing the data to the media. The function may write less data than requested if it is writing to a slow device such as a terminal and the task catches a signal.

ERRORS REPORTED

EACCES	The file descriptor references a file that isn't open for writing
EBADF	The file descriptor is out of range or does not reference an open file
EINTR	The task caught a signal and that caused this function to end abnormally
EIO	The operating system reports an I/O error
ENOSPC	The device is full
EPIPE	Attempting to write to a broken pipe

NOTES

This operation is most efficient when <bufad> and <nbytes> are evenly divisable by 512.

A broken pipe is one which has been closed for reading.

SEE ALSO

C Library: fwrite()

System Call: creat(), dup(), dup2(), open(), pipe(), read()

SPECIAL SUPPORT LIBRARIES

The 4404 comes with several special libraries to support the 4404 hardware. These libraries may be used with programs written either in C or 68000 assembly language. The functions in the libraries conform to C calling conventions.

THE 'C' LIBRARY

The standard 'C' library, "clib" which contains the standard 'C' functions for the the 4404 resides in the directory "/lib". The following 'C' functions are available:

abs

Absolute value function.

SYNOPSIS

```
int abs(i)
int       i;
```

Arguments

i¦ Value whose absolute value is to be calculated

Returns

The absolute value of the argument <i>

DESCRIPTION

This function calculates the absolute value of the argument <i>. It returns the calculated value as its result.

NOTES

If <i> is the largest negative number, "abs()" returns that value as its result.

asctime

Generate a time stamp.

SYNOPSIS

```
#include <time.h>
char *asctime(dttm)
struct tm     *dttm;
```

Arguments

dttm¦ The address of a structure containing date and time information

Returns

The address of the generated time stamp

DESCRIPTION

This function generates a time stamp representing the date and time information in the structure reference by <dttm>. It returns the address of the time stamp as its result.

A time stamp is a 26-character string of characters (including the terminating null-character) consisting of the day of the week, the month of the year, the day of the month, the hour, minute, second, and year. The time stamp is generated by the "sprintf()" format:

"%3s %3s %2.2d %2.2d:%2.2d:%2.2d %4.4d n"

NOTES

The character-string referenced by the result of this function is in static memory and is overwritten by subsequent calls to this function and "ctime()".

SEE ALSO

C Library: ctime(), gmtime(), localtime(), sprintf()

System Call: time()

Command: date

_atoh

Convert a string of hexadecimal characters to an integer.

SYNOPSIS

```
long _atoh(str)
char     *str;
```

Arguments

str¦ The address of the character-string to convert

Returns

The integer generated from the character-string referenced by <str>

DESCRIPTION

This function generates a long from the character-string referenced by <str>. It returns that value as its result. The function expects the character-string to contain optional whitespace (see "isspace()"), which is ignored, followed optionally by a ('0') and an ('x') or ('X'), which are ignored, followed by a string of hexadecimal digits (see "isxdigit()"). It continues converting until it reaches the end of the string or it finds inappropriate character.

NOTES

The function ignores overflow errors. The conversion is performed by strtol(str, (char **) NULL, 16)

SEE ALSO

C Library: atoi(), _atoo(), atol(), _atos(), strtol()

atoi

Convert a string of decimal characters to an integer.

SYNOPSIS

```
int atoi(str)
char     *str;
```

Arguments

str The address of the character-string to convert

Returns

The integer generated from the character-string referenced by <str>

DESCRIPTION

This function generates an int from the character-string referenced by <str>. It returns the generated value as its result. The function expects the character-string to contain optional whitespace (see "isspace()"), which is ignored, followed by a string of decimal digits (see "isdigit()"). It continues converting until it reaches the end of the string or it finds an inappropriate character.

NOTES

Overflow errors are ignored. The conversion is performed by (int) strtol(str, (char **) NULL, 10)

SEE ALSO

C Library: _atoh(), _atoo(), atol(), _atos(), strtol()

atol

Convert a string of decimal characters to an integer.

SYNOPSIS

```
long atol(str)
char     *str;
```

Arguments

str¦ The address of the character-string to convert

Returns

The integer generated from the character-string referenced by <str>

DESCRIPTION

This function generates a long from the character-string referenced by <str>. It returns that value as its result. The function expects the character-string to contain optional whitespace (see "isspace()"), which is ignored, followed by a string of decimal digits (see "isdigit()"). The function converts until it reaches the end of the string or it detects an inappropriate character.

NOTES

Overflow errors are ignored. The conversion is performed by strtol(str, (char **) NULL, 10)

SEE ALSO

C Library: _atoh(), atoi(), _atoo(), _atos(), strtol()

_atoo

Convert a string of octal characters to an integer.

SYNOPSIS

```
long _atoo(str)
char     *str;
```

Arguments

str¦ The address of the character-string to convert

Returns

The integer generated from the character-string referenced by <str>

DESCRIPTION

This function generates a long from the character-string referenced by <str>. It returns that value as its result. The function expects the character-string to contain optional whitespace (see "isspace()"), which is ignored, followed by a string of octal digits ([0-7]). The function continues until it reaches the end of the string or it finds an inappropriate character.

NOTES

Overflow errors are ignored. The conversion is performed by strtol(str, (char **) NULL, 8)

SEE ALSO

C Library: _atoh(), atoi(), atol(), _atos(), strtol()

_atos

Convert a string of decimal characters to an integer.

SYNOPSIS

```
short _atos(str)
char    *str;
```

Arguments

str¦ The address of the character-string to convert

Returns

The integer generated from the character-string referenced by <str>

DESCRIPTION

THis function generates a short from the character-string referenced by <str>. It returns that value as its result. The function expects the character-string to contain optional whitespace (see "isspace()"), which is ignored, followed by a string of decimal digits (see "isdigit()"). The function converts until it reaches the end of the string or it finds an inappropriate character.

NOTES

Overflow errors are ignored. The conversion is performed by (short) strtol(str, (char **) NULL, 10).

SEE ALSO

C Library: _atoh(), atoi(), atol(), _atoo(), strtol()

calloc

Allocate memory.

SYNOPSIS

```
char *calloc(num, size)
unsigned  num;
unsigned  size;
```

Arguments

num	The number of units to allocate
size	The size of a unit

Returns

The address of the allocated block of memory or (char*) NULL if no memory is available

DESCRIPTION

This function allocates <num>*<size> bytes of memory from the arena of available memory. It returns the address of the first byte of the allocated memory or (char*) NULL if no memory is available. The first byte of the allocated memory is aligned for any use.

NOTES

Return allocated memory to the arena of available memory by using "free()".

SEE ALSO

C Library: free(), malloc(), realloc()

System Call: brk(), cdata(), sbrk()

clearerr

Clear the stream's error-indicator.

SYNOPSIS

```
#include <stdio.h>
int clearerr(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream

Returns

Undefined

DESCRIPTION

This function clears (resets) the error-indicator on the standard I/O stream <stream>.

NOTES

This function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once. The function "ferror()" tests a stream's error-indicator.

SEE ALSO

C Library: fdopen(), ferror(), fopen(), stderr, stdin, stdout

_crypt

Encrypt a character-string.

SYNOPSIS

```
void _crypt(crypw, pw)
char     *crypw;
char     *pw;
```

Arguments

crypw	The address of the target buffer to get the encrypted string
pw	The address of the character-string to be encrypted

Returns

Void

DESCRIPTION

This function encrypts the first eight characters of the character-string referenced by <pw> using the standard UniFLEX encryption algorithm, generating a character-string containing sixteen characters. It copies the generated character-string to the the target buffer referenced by <crypw>.

If <pw> references a null-string, it copies a null-string to the target buffer referenced by <crypw>.

NOTES

The function produces unpredictable results if the character-string's length is greater than eight characters.

SEE ALSO

Command: password

ctime

Generate a time stamp.

SYNOPSIS

```
#include <time.h>
char *ctime(pclock)
long     *pclock;
```

Arguments

pclock¦ The address of the system-time value

Returns

The address of the generated time stamp

DESCRIPTION

This function generates a time stamp representing the date and time in the local time zone from the system time value referenced by <pclock>. It returns the address of the generated time stamp.

A time stamp is a 26-character string of characters (including the terminating null-character) consisting of the day of the week, the month of the year, the day of the month, the hour, minute, second, and year. The time stamp is generated by the "sprintf()" format:

"%3s %3s %2.2d %2.2d:%2.2d:%2.2d %4.4d n"

NOTES

The result of this function is in static memory. Subsequent calls to "ctime()" or "asctime()" overwrite that memory. This function calls "tzset()" which sets up "daylight", "timezone", and "tzname". A system time value is a long containing the number of seconds since the epoch. The epoch is 00:00 GMT (midnight) on January 1, 1980. If the system time value referenced by <pclock> is less than "timezone", "ctime()" will generate a time stamp for 00:00 on Janu- ary 1, 1980, locally.

SEE ALSO

C Library: asctime(), daylight, gmtime(), localtime(), sprintf(), timezone, tzname, tzset()

System Call: time

Command: date

daylight

Daylight savings time flag.

SYNOPSIS

```
#include <time.h>
extern int daylight;
```

DESCRIPTION

This variable is non-zero if and only if U. S. A. Standard Daylight Savings Time is being observed and should be applied to all conversions of time to be expressed in the local time zone. Otherwise, it is zero.

This variable is initialized automatically by "localtime()" and "ctime()" and may be initialized explicitly by "tzset()". The value is zero before initialization.

SEE ALSO

C Library: ctime(), localtime(), timezone, tzname, tzset()

System Call: ftime()

endpwent

End password-file handling.

SYNOPSIS

```
#include <pwd.h>
void endpwent();
```

Arguments

None

Returns

Void

DESCRIPTION

This function ends password-file handling initiated by "getpwent()", "getpwnam()", or "getpwuid()". It frees the resources allocated to and closes the files opened by those routines.

NOTES

This function does nothing if "getpwent()", "getpwnam()", or "getpwuid()" has not been called or "endpwent()" has been called since the last call to one of these functions.

SEE ALSO

C Library: getpwent(), getpwnam(), getpwuid(), setpwent()

endutent

End multi-user login-file handling.

SYNOPSIS

```
#include <utmp.h>
void endutent();
```

Arguments

None

Returns

Void

DESCRIPTION

This function ends multi-user login-file handling initiated by "getutent()" or "getutline()". It frees the resources allocated to and closes the files opened by those routines.

NOTES

This function does nothing if neither "getutent()" nor "getutline()" has been called or "endutent()" has been called since the last call to one of these functions.

SEE ALSO

C Library: getutent(), getutline(), setutent()

_exit

Exit the program.

SYNOPSIS

```
void _exit(code)
int  code;
```

Arguments

code	The value to give to the operating system to use as the task-termination code

Returns

Void

DESCRIPTION

This function ends execution of the program by terminating the task, giving <code> to the operating system to use as the task-termination code.

NOTES

This function does not return to the caller. The standard I/O streams open at the call to this function will be closed but the any data in buffered streams opened for writing will not be flushed to the attached file. To flush these buffers, exit the program using "exit()".

SEE ALSO

C Library: exit()

System Call: fork(), wait()

fclose

Close a stream.

SYNOPSIS

```
#include <stdio.h>
int fclose(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream to close

Returns

Zero if successful, EOF otherwise

DESCRIPTION

This function closes the standard I/O stream <stream> and frees any resources which were automatically allocated to the stream. If the stream is opened for writing and is buffered, it flushes any buffered data to the associated file.

The function returns EOF if it encounters an error while closing the stream, otherwise it returns zero.

SEE ALSO

C Library: fdopen(), fflush(), freopen(), fopen(), stderr, stdin, stdout

System Call: close(), open()

fdopen

Attach an open file to a stream.

SYNOPSIS

```
#include <stdio.h>
FILE *fdopen(fildes, mode)
int       fildes;
char     *mode;
```

Arguments

fildes¦	A file descriptor for the file to attach
mode¦	The address of a character-string describing the requested open mode

Returns

The standard I/O stream to which the open file has been attached, or (FILE *) NULL if the function detected an error

DESCRIPTION

This function attaches the file referenced by the file descriptor <fildes> to a standard I/O stream. An open file descriptor is returned by the system-call functions "creat()", "dup()", "dup2()", "open()", and "pipe()". Valid open modes are "r", "w", or "a" for read, write, and append access. Read and write access begins at the current position in the file, append access begins at the end of the file. This function is typically used to permit standard I/O functions on a file opened by some means other than the standard I/O function "fopen()".

The function returns the standard I/O stream to which the file has been attached, or (FILE *) NULL if there was an error. Possible errors include a bad file descriptor <fildes>, an unknown open mode, or attempting to exceed the maximum open-stream limit.

NOTES

The access mode is suppose to match the open mode of the file. This is not currently checked since there is no way to coax the open mode from the operating system given an open file number. This function does not yet support the UNIX System V access modes of "r+", "w+", or "a+".

Files attached to streams using this routine should be closed using "fclose()" to ensure that the resources automatically allocated to the stream are released to the system and that any data gets flushed.

SEE ALSO

C Library: fclose(), freopen(), fopen()

System Call: close(), dup(), dup2(), open(), pipe()

feof

Test a stream's end-of-file indicator.

SYNOPSIS

```
#include <stdio.h>
int feof(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream

Returns

Non-zero if the end-of-file indicator on the stream is set (on); zero otherwise.

DESCRIPTION

This function tests the end-of-file indicator on the standard I/O stream <stream>. It returns a non-zero value if the indicator is set, other- wise it returns zero.

A standard I/O function sets a stream's end-of-file indicator when it attempts to read data from the stream produce no data and no errors.

NOTES

This function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once.

SEE ALSO

C Library: fdopen(), ferror(), fopen(), stderr, stdin, stdout

ferror

Test a stream's error-indicator.

SYNOPSIS

```
#include <stdio.h>
int ferror(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream

Returns

Non-zero if the error-indicator on the stream is set (on); zero otherwise.

DESCRIPTION

This function tests the error-indicator on the standard I/O stream <stream>. It returns a non-zero value if the indicator is set, otherwise it returns zero.

A standard I/O function sets a stream's error-indicator if it attempts to perform I/O on the stream and the operating system reports an error. The function "clearerr()" clears a stream's error-indicator.

NOTES

This function is implemented as a macro. Macro side-effects are not possible since the macro references its argument only once.

SEE ALSO

C Library: clearerr(), feof(), fdopen(), fopen(), stderr, stdin, stdout

fflush

Flush a stream opened for write access.

SYNOPSIS

```
#include <stdio.h>
int fflush(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream to flush

Returns

Zero if successful, EOF otherwise

DESCRIPTION

This function flushes any buffered data written to the standard I/O stream <stream>. The stream must be opened for write or append access. The function returns EOF if it encounters an error flushing the stream, otherwise it returns zero.

SEE ALSO

C Library: fclose(), fdopen(), freopen(), fopen(), stderr, stdin, stdout

fgetc

Read a character from a stream.

SYNOPSIS

```
#include <stdio.h>
int fgetc(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream to read from

Returns

The character read if successful, otherwise EOF

DESCRIPTION

This function reads the next character from the standard I/O stream <stream>. If it succeeded, it returns that character as its result, cast into an int with no sign extension, otherwise it returns EOF.

NOTES

The character read is considered to be an unsigned char so there is no sign extension when converting the character to an integer value for returning.

SEE ALSO

C Library: fdopen(), fopen(), fputc(), fread(), getc(), getchar(), stdin

fgets

Read a character-string from a stream.

SYNOPSIS

```
#include <stdio.h>
char *fgets(ptr, count, stream)
char     *ptr;
int       count;
FILE     *stream;
```

Arguments

ptr|	The address of the target buffer
count|	The size of the target buffer
stream|	The standard I/O stream to read from

Returns

The argument <ptr> if successful, (char*) NULL otherwise

DESCRIPTION

This function reads characters from the standard I/O stream <stream> un til it reads <count>-1 characters, it reads an end-of-line character, or it reaches the end of the file. It writes these characters to the buffer whose address is <ptr>. It appends a null-character (' 0') onto the characters read, making a character-string, then returns <ptr> as its result.

If it detects an error, the function returns (char*) NULL and does not alter the target buffer.

SEE ALSO

C Library: fdopen(), fgetc(), fopen(), fputs(), gets(), stdin

fileno

Get a file descriptor for the file attached to a stream.

SYNOPSIS

```
int fileno(stream)
FILE     *stream;
```

Arguments

stream¦ A standard I/O stream

Returns

A file descriptor for the file attached to the stream

DESCRIPTION

This function returns a file descriptor for the file attached to the stream <stream>. This file descriptor can be used by various system-call functions, such as "read()", and "write()".

NOTES

The function's results are undefined if <stream> does not reference an open stream.

SEE ALSO

C Library: fdopen(), fopen(), stderr, stdin, stdout

System Call: dup(), dup2(), open(), read(), write()

fopen

Open a file and attach it to a standard I/O stream.

SYNOPSIS

```
#include <stdio.h>
FILE *fopen(pathnam, mode)
char     *pathnam;
char     *mode;
```

Arguments

pathnam¦ The address of a character-string containing a path-name to the file to open

mode¦ The address of a character-string containing the open mode

Returns

If successful, the stream to which the open file has been attached, otherwise (FILE *) NULL

DESCRIPTION

This function opens the file reached by the pathname in the character- string referenced by <pathnam>. The character-string referenced by <mode> describes to the function the access type desired by the program. The function then attaches the open file to a standard I/O stream.

If the function succeeds, it returns the standard I/O stream as its result. Otherwise, it returns (FILE *) NULL. The function fails if If the operating system reports an error, the program has the maximum number of streams open, or the open mode is not valid. If the operating system reports an error, "errno" will contain the system error code.

The open mode describes the type of access requested for the file. Valid open modes are "r", "w", or "a", for read, write, or append access, respectively.

If the open mode is "r", the function opens the file for reading if the file already exists, setting the current position at the beginning of the file. If the pathname <pathnam> doesn't reach a file, the function fails.

If the open mode is "w", the function opens the file for writing. If the file already exists, the function truncates the file to a length of zero. Otherwise, it creates a file with a length of zero. It sets the current position at the beginning of the file.

If the open mode is "a", the function opens the file for writing. If the file doesn't exist, the function creates a file with a length of zero. It sets the current position at the end of the file.

NOTES

These open modes are not currently supported: "r+", "w+", "a+". The include-file "<stdio.h>" defines the data type "FILE". This data type is a structure containing all of the information about an open stream. For brevity, this and other manual pages discuss a pointer to the data type "FILE" as simply a "stream", instead of calling it a pointer to a structure defining the characteristics of a stream.

SEE ALSO

C Library: fclose(), fdopen(), fgetc(), fgets(), fputc(), fputs(), fread(), freopen(), fwrite()

System Call: close(), open()

fputc

Write a character to a stream.

SYNOPSIS

```
#include <stdio.h>
int fputc(c, stream)
char      c;
FILE     *stream;
```

Arguments

c¦	The character to write
stream¦	The standard I/O stream to write to

Returns

The value written if successful, EOF otherwise.

DESCRIPTION

This function writes the character <c> to the standard I/O stream <stream>. The function returns the character written as its result if it successfully writes the character to the stream, otherwise, it re- turns EOF.

NOTES

If the stream is buffered but not line-buffered, standard I/O does not write the character to the attached file until the stream's buffer is full or the stream is closed. If the stream is line-buffered, standard I/O does not write the char- acter to the attached file until one of the following conditions: an end-of-line character (EOL) is written to the stream, a standard I/O function attempts to read data from a terminal, the stream's buffer is full, or the stream is closed.

If the function succeeds, it returns the value of the char argument <c> converted to int as though <c> were an unsigned char.

SEE ALSO

C Library: fdopen(), fgetc(), fopen(), fputs(), putc(), putchar()

fputs

Write a character-string to a stream.

SYNOPSIS

```
#include <stdio.h>
int fputs(s, stream)
char     *s;
FILE     *stream;
```

Arguments

s¦ The address of the character-string to write to the stream

stream¦ The standard I/O stream to write to

Returns

Zero if successful, EOF otherwise

DESCRIPTION

This function writes the characters in character-string referenced by <s> to the standard I/O stream <stream>. The function returns zero as its result if it successfully writes the characters to the stream, otherwise it returns EOF.

NOTES

The function does not write to the stream the null-character terminating the character-string.

If the stream is buffered but not line-buffered, standard I/O does not write the characters to the attached file until it fills the stream's buffer or closes the stream.

If the stream is line-buffered, standard I/O does not write the character to the attached file until it writes an end-of-line character (EOL) to the stream, attempts to read data from a terminal, fills the stream's buffer, or closes the stream.

SEE ALSO

C Library: fdopen(), fgets(), fopen(), fputc(), puts()

fread

Read data from a stream.

SYNOPSIS

```
#include <stdio.h>
int fread(ptr, size, count, stream)
char     *ptr;
int       size;
int       count;
FILE     *stream;
```

Arguments

ptr¦	Address of the buffer to contain the data read
size¦	The size of an item to read
count¦	The maximum number of items to read
stream¦	The standard I/O stream

Returns

The number of complete items read, if any

DESCRIPTION

This function reads at most <count> items of <size> bytes from the I/O stream <stream>, placing the data read into the buffer whose address is <ptr>. The function reads data until it reads the requested number of data items, reaches the end of the file, or detects an error on the input stream. The function returns as its result the number of complete items read from the stream.

NOTES

If the function reaches the end of the file or encounters an error while reading a data item, it writes that partial item to the target buffer but does not count that partially read item in the count of items read, which it returns as its result. The target buffer needs no special boundary alignment. If <count> is less than or equal to zero, the function does not attempt to read any data and returns zero as its result.

SEE ALSO

C Library: fdopen(), fopen(), fwrite()

System Call: read(), write()

free

Free a block of allocated memory.

SYNOPSIS

```
void free(ptr)
char     *ptr;
```

Arguments

ptr¦ The address of the block of memory to free

Returns

Void

DESCRIPTION

This function returns the block of memory whose address is <ptr> to the arena of available memory. The block of memory must have been allocated by "malloc()", "calloc()", or "realloc()".

NOTES

If the argument <ptr> is the address of a block that has already been freed, or is some value other than one returned by "malloc()", "calloc()", or "realloc()", the function corrupts the arena of available memory and makes subsequent calls to "malloc()", "calloc()", "realloc()", and "free()" behave unpredictably.

SEE ALSO

C Library: calloc(), malloc(), realloc()

System Call: brk(), cdata(), sbrk()

freopen

Reopen an open stream.

SYNOPSIS

```
#include <stdio.h>
FILE *freopen(pathnam, mode, stream)
char     *pathnam;
char     *mode;
FILE     *stream;
```

Arguments

pathnam¦ The address of a character-string containing a path-name to the file to open and attach to the stream

mode¦ The address of a character-string containing the open mode

stream¦ The standard I/O stream to reopen

Returns

The argument <stream> if successful, (FILE *) NULL otherwise

DESCRIPTION

This function closes the standard I/O stream <stream>, opens the file reached by the pathname in the character string referenced by <pathnam>, with the open mode specified by the character-string referenced by <mode>, and attaches the newly opened file to the stream.

If the function succeeds, it returns the standard I/O stream <stream> as its result. Otherwise, it returns (FILE *) NULL. The function fails if If the operating system reports an error, the stream is not open, the program has the maximum number of streams open, or the open mode is not valid. If the operating system reports an error, "errno" will contain the system error code.

The open mode describes the type of access requested for the file. Valid open modes are "r", "w", or "a", for read, write, or append access, respectively.

If the open mode is "r", the function opens the file for reading if the file already exists, setting the current position at the beginning of the file. If the pathname <pathnam> doesn't reach a file, the function fails.

If the open mode is "w", the function opens the file for writing. If the file already exists, the function truncates the file to a length of zero. Otherwise, it creates a file with a length of zero. It sets the current position at the beginning of the file.

If the open mode is "a", the function opens the file for writing. If the file doesn't exist, the function creates a file with a length of zero. It sets the current position at the end of the file.

NOTES

This function is typically used to attach files to automatically opened streams, such as "stdin", "stdout", and "stderr". The file that was originally attached to the stream <stream> is closed without regard to the eventual outcome of the function call.

SEE ALSO

C Library: fclose(), fdopen(), fopen(), stderr, stdin, stdout

System Call: close(), open()

fscanf

Read and interpret formatted data from a stream.

SYNOPSIS

```
#include <stdio.h>
int fscanf(stream, format [, addrlist])
FILE     *stream;
char     *format;
```

Arguments

stream	The standard I/O stream to read from
format	The address of a character-string containing a format description

Returns

The number of items in the address-list <addrlist> that it successfully assigns or EOF if an error occurs before it assigns any data

DESCRIPTION

This function reads and interprets data from the standard I/O stream <stream> according to the format description in the character-string referenced by <format>.

fseek

Reposition a stream.

SYNOPSIS

```
#include <stdio.h>
int fseek(stream, offset, type)
FILE     *stream;
long      offset;
int       type;
```

Arguments

stream	The standard I/O stream to reposition
offset	A value indicating the desired position, in bytes
type	A value indicating type of positioning

Returns

Zero if the positioning was successful, EOF otherwise.

DESCRIPTION

This function changes the current offset into the stream referenced by <stream>. If <type> is 0, the value <offset> is a byte offset from the beginning of the stream. If <type> is 1, the value <offset> is a byte offset from the current position in the stream. If <type> is 2, the value <offset> is a byte offset from the end of the stream.

The function returns zero if it successfully repositioned the stream, otherwise it returns EOF.

NOTES

- If the function is not successful, "errno" will contain the UniFLEX error code indicating the error.
- A file may be extended by requesting a seek relative to the end of the file with a positive offset.
- A file may not be positioned before its beginning. Calling this function undoes any effect of "ungetc()".
- A stream attached to terminal may not be repositioned.

SEE ALSO

C Library: fdopen(), fopen(), ftell(), rewind()

System Call: lseek()

ftell

Get the current position of a stream.

SYNOPSIS

```
long ftell(stream)
FILE     *stream;
```

Arguments

stream¦ A standard I/O stream

Returns

The current position of the stream, in bytes

DESCRIPTION

This function examines the standard I/O stream <stream>, determines its current position relative to the beginning of the stream, and returns a value indicating that position.

If the stream is opened for read access, the current position contains the next character that is read. If the stream is opened for write or append access, the current position is where the next character is written.

NOTES

This function is not affected by a character pushed onto the stream by "ungetc()".

This function will takes into account I/O buffering, which means it may return a different position than the UniFLEX system call "lseek()".

SEE ALSO

C Library: fdopen(), fopen(), fseek(), rewind()

System Call: lseek()

fwrite

Write data to a stream.

SYNOPSIS

```
#include <stdio.h>
int fwrite(ptr, size, count, stream)
char     *ptr;
int       size;
int       count;
FILE     *stream;
```

Arguments

ptr¦	The address of the buffer containing the data to write
size¦	The size of an item to write
count¦	The number of items to write
stream¦	The standard I/O stream to write data to

Returns

The number of complete items written, if any

DESCRIPTION

This function writes <count> items of <size> bytes from the buffer whose address is <ptr> to the standard I/O stream <stream>. The function writes data until it writes the requested number of data items, or it detects an I/O error.

The function returns as its result the number of complete items written to the stream.

NOTES

- The data buffer whose address is <ptr> needs no special boundary alignment.
- If <count> is less than or equal to zero, the function does not attempt to write any data and returns zero as its result.
- If the stream is buffered but not line-buffered, standard I/O does not write the character to the attached file until the stream's buffer is full or the stream is closed.
- If the stream is line-buffered, standard I/O does not write the character to the attached file until an end-of-line character (EOL) is written to the stream, a standard I/O function attempts to read data from a terminal, the stream's buffer is full, or the stream is closed.

SEE ALSO

C Library: fdopen(), fopen(), fread()

System Call: read(), write()

getc

Read a character from a stream.

SYNOPSIS

```
#include <stdio.h>
int getc(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream to read from

Returns

The character read if successful, otherwise EOF.

DESCRIPTION

This function reads the next character from the standard I/O stream <stream>. If it succeeded, it returns that character as its result, cast into an int with no sign extension, otherwise it returns EOF.

NOTES

The character read is considered to be an unsigned char so there is no sign extension when converting the character to an integer value for returning. This function is exactly like "fgetc()" and is included for compatability with other systems.

SEE ALSO

C Library: fdopen(), fgetc(), fopen(), fputc(), fread(), getchar(), stdin

getchar

Read a character from the standard input stream.

SYNOPSIS

```
#include <stdio.h>
int getchar()
```

Arguments

None

Returns

The character read if successful, otherwise EOF

DESCRIPTION

This function reads the next character from the standard I/O stream "stdin". If it succeeded, it returns that character as its result, cast into an int with no sign extension, otherwise it returns EOF.

NOTES

The character read is considered to be an unsigned char so there is no sign extension when converting the character to an integer value for returning.

SEE ALSO

C Library: fdopen(), fgetc(), fopen(), fputc(), fread(), getc(), stdin

getcwd

Get the pathname of the working directory.

SYNOPSIS

```
char *getcwd(ptr, size)
char     *ptr;
int       size;
```

Arguments

ptr¦ The address of the buffer to receive the pathname of the working directory, or (char *) NULL

size¦ Size, in bytes, of the target buffer

Returns

The address of the character-string containing the pathname to the working directory

DESCRIPTION

This function generates a character-string containing the complete path-name of the working directory. If the length of that string is greater than <size>, the function returns (char *) NULL. If <ptr> is equal to (char *) NULL, the function allocates a buffer using "malloc()", copies the generated character-string into the allocated buffer, and returns as its result the address of the allocated buffer. Otherwise, it copies the generated character-string into the buffer whose address is <ptr> and returns <ptr> as its result.

NOTES

The function returns (char *) NULL if <ptr> is (char *) NULL and "malloc()" is unable to allocate <size> bytes of memory. If <ptr> is (char *) NULL, the buffer allocated by "getcwd()" may be freed using "free()". This function generates the complete pathname of the working directory, beginning at root of the device containing the directory. Someday, the complete pathname will begin at the root directory on the root device.

SEE ALSO

C Library: malloc(), free()

System Call: chdir()

Command: path

getpass

Get a password using a prompt.

SYNOPSIS

```
char *getpass(prompt)
char     *prompt;
```

Arguments

prompt¦ The address of the character-string containing the prompt

Returns

The address of a character-string containing the password read, or (char *) NULL if there was an error

DESCRIPTION

This function writes the characters in the character-string referenced by <prompt> to the standard I/O output stream "stderr".

- It clears the echo attribute on the terminal associated with the standard I/O input stream "stdin", then reads characters from "stdin" up to the first end-of-line character (EOL) or to the end of the file.
- It saves the first eight characters in a static buffer, discarding the remaining characters, if any, and the end-of-line character, if any.
- It restores the echo attribute on the terminal to its original state, terminates the characters saved with a null-character, completing the character-string, then returns the address of that character-string as its result.
- If the function encounters an error, it restores the terminal to its original state and returns (char *) NULL as its result.

NOTES

This function uses standard I/O and may enlarge a program more than expected.

Nothing is written to "stderr" if <prompt> is (char *) NULL. The function catches keyboard, quit, alarm, and hang-up signals. If it catches a signal, it resets the terminal to its original configuration and then resignals the signal so the calling program may handle that signal. If the function returns indicating an error, "errno" contains the system error code.

The character-string referenced by the result of this function is in static memory and is overwritten by subsequent calls to this func- tion.

The standard I/O output stream "stderr" must be attached to a terminal unless <prompt> is (char *) NULL. Otherwise, the function will return (char *) NULL with "errno" set to ENOTTY. The standard I/O input stream "stdin" must be attached to a terminal or the function will return (char *) NULL with "errno" set to ENOTTY.

SEE ALSO

C Library: fputs(), gets(), stderr, stdin

getpw

Get a password-file entry based on a user-ID.

SYNOPSIS

```
int getpw(uid, ptr)
int       uid;
char     *ptr;
```

Arguments

uid¦	The user-ID number to search for
ptr¦	The buffer to contain the record found

Returns

Zero if a record was successfully found, EOF otherwise.

DESCRIPTION

This function searches the system's password-file for the first correctly formatted record with a user-ID field equivalent to <uid>. If one is found, it copies that record, including the terminating end-of-line character (EOL), into the buffer whose address is <ptr> and returns zero as its result. Otherwise, it leaves the buffer whose address is <ptr> unchanged and returns EOF as its result.

NOTES

This function is obsolete but was included for compatability with older systems. New applications should use "getpwuid()".

The caller is responsible for ensuring that the buffer whose address is <ptr> is large enough to hold the data. This function uses standard I/O and may make the calling program larger than expected. The system's password file is "/etc/log/password".

SEE ALSO

C Library: getpwent(), getpwuid(),

Command: password

getpwent

Get and decode the next entry in the system password file.

SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwent();
```

Arguments

None

Returns

The address of the structure containing information from the record read, or (struct passwd *) NULL if no record was read

DESCRIPTION

This function reads and decodes the next correctly formatted entry in the system password file. The information is saved in a static structure (defined below) and the address of that structure is returned as the its result. If it couldn't read a record from the system password file, the function returns (struct passwd *) NULL as its result.

If no previous "getpwent()", "getpwnam()", or "getpwuid()" has been successfully attempted, or "endpwent()" has been called since the last call to "getpwent()", "getpwnam()", or "getpwuid()" this function opens the system password file and positions it to the first record in the file. After the function completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The function "endpwent()" closes the system password file. Task termination also closes the file. The function "setpwent()" rewinds the system password file, positioning it to the first record of the file.

The include-file "<pwd.h>" defines structures and constants used when manipulating the data in the system password file. The format of the struct passwd structure referenced by the result of this function is as follows:

```
struct passwd
{
char     *pw_name;
char     *pw_passwd;
int       pw_uid;
char     *pw_dir;
char     *pw_shell;
};
```

Where:

- "pw_name" is the address of a character-string containing the user-name.
- "pw_passwd" is the address of a character-string containing the encrypted password.
- "pw_uid" contains the user's identifying number (user-ID).
- "pw_dir" is the address of a character_string containing the user's home directory.
- "pw_shell" is the address of a character- string containing the shell-command for the first program to run after logging on.

A null-string as the encrypted password indicates that the user has no password, and a null-string as the shell-command indicates that the initial program is the standard shell.

NOTES

The structure referenced by the result of this function and the character-strings referenced by the values in that structure are in static memory and are overwritten by subsequent calls to "getpwent()", "getpwnam()", and "getpwuid()".

The function ignores improperly formatted records in the system password file. This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The function will return (struct passwd *) NULL if the user does not have permission to access the password file, the current position on the system password file is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another. The system password file is "/etc/log/password".

SEE ALSO

C Library: endpwent(), getpw(), getpwnam(), getpwuid(), putpwent(), setpwent()

Command: password

getpwnam

Get and decode the next entry in the system password file containing the given user-name.

SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwnam(name)
char     *name;
```

Arguments

name| The address of a character-string containing the user-name

Returns

The address of the structure containing the information in the record read, or (struct passwd *) NULL if no record was read

DESCRIPTION

This function reads and decodes the next correctly formatted entry in the system password file that contains a user-name matching that in the character-string referenced by the argument <name>. The information is saved in a static structure (defined below) and the address of that structure is returned as the its result. If it couldn't find a record in the system password file containing the specified user-name, the function returns (struct passwd *) NULL as its result.

If no previous "getpwent()", "getpwnam()", or "getpwuid()" has been successfully attempted, or "endpwent()" has been called since the last call to "getpwent()", "getpwnam()", or "getpwuid()" this function opens the system password file and positions it to the first record in the file. After the function completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The function "endpwent()" closes the system password file. Task termination also closes the password file. The function "setpwent()" rewinds the system password file.

The include-file "<pwd.h>" defines constants and structures used when manipulating entries in the system password file. The format of the struct passwd structure referenced by the result of this function is as follows:

```
struct passwd
{
char     *pw_name;
char     *pw_passwd;
int       pw_uid;
char     *pw_dir;
char     *pw_shell;
};
```

The entry "pw_name" is the address of a character-string containing the user-name, "pw_passwd" is the address of a character-string containing the encrypted password, "pw_uid" contains the user's identifying number (user-ID), "pw_dir" is the address of a character-string containing the user's initial home-directory, and "pw_shell" is the address of a character-string containing the shell-command for the first program to run after logging on.

A null-string as the encrypted password indicates that the user has no password. A null-string as the shell-command indicates that the initial program is the standard shell.

NOTES

The structure referenced by the result of this function and the character-strings referenced by the values in that structure are in static memory and are overwritten by subsequent calls to "getpwent()", "getpwnam()", and "getpwuid()".

The function ignores improperly formatted records. This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The function will return (struct passwd *) NULL if the user does not have permission to access the password file, if the current position on the system password file is end-of-file, or if the user has the maximum number of standard I/O streams open and can not open anoth- er. The system password file is "/etc/log/password".

SEE ALSO

C Library: endpwent(), getpw(), getpwent(), getpwuid(), putpwent(), setpwent()

Command: password

getpwuid

Get and decode the next entry in the system password file containing the given user-ID number.

SYNOPSIS

```
#include <pwd.h>
struct passwd *getpwuid(uid)
int       uid;
```

Arguments

uid¦ The user-ID number to search for

Returns

The address of the structure containing the information in the record read, or (struct passwd *) NULL) if no record was read

DESCRIPTION

This function reads and decodes the next correctly formatted entry in the system password file that contains a user-ID number matching the user-ID number <uid>. The information is saved in a static structure (defined below) and the address of that structure is returned as the its result. If it couldn't find a record in the system password file con- taining the specified user-ID number, the function returns (struct passwd *) NULL as its result.

If no previous "getpwent()", "getpwnam()", or "getpwuid()" has been successfully attempted, or "endpwent()" has been called since the last call to "getpwent()", "getpwnam()", or "getpwuid()" this function opens the system password file and positions it to the first record in the file. After the function completes, the system password file remains open and is positioned to the record immediately following the record read, or to the end of the file if no record was successfully read.

The function "endpwent()" closes the system password file. Task termination also closes the file. The function "setpwent()" rewinds the system password file, positioning it to the beginning of the first record in the file.

The include-file "<pwd.h>" defines structures and constants used when reading and manipulating entries in the system password file. The format of the struct passwd structure referenced by the result of this function is:

```
struct passwd
{
char     *pw_name;
char     *pw_passwd;
int       pw_uid;
char     *pw_dir;
char     *pw_shell;
};
```

The entry "pw_name" is the address of a character-string containing the user-name, "pw_passwd" is the address of a character-string containing the encrypted password, "pw_uid" contains the user's identifying number (user-ID), "pw_dir" is the address of a character-string containing the user's initial home-directory, and "pw_shell" is the address a character-string containing the shell-command for the first program to run after logging on.

A null-string as the encrypted password indicates that the user has no password, and a null-string as the shell-command indicates that the initial program is the standard shell.

NOTES

The structure referenced by the result of this function and the character-strings referenced by the values in that structure are in static memory and will be overwritten by subsequent calls to "getpwent()", "getpwnam()", and "getpwuid()".

Improperly formatted records in the system password file are ignored.

This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The function will return (struct passwd *) NULL if the user does not have permission to access the password file, the current position on the system password file is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another. The system password file is "/etc/log/password".

SEE ALSO

C Library: endpwent(), getpw(), getpwent(), getpwnam(), putpwent(), setpwent()

Command: password

gets

Read a character-string from the standard input stream.

SYNOPSIS

```
#include <stdio.h>
char *gets(ptr)
char     *ptr;
```

Arguments

ptr¦ The address of the target buffer

Returns

The argument <ptr> if successful, (char *) NULL otherwise

DESCRIPTION

This function reads characters from the standard I/O input stream "stdin" until it reads an end-of-line character, or reaches the end of the file. It places the characters in the buffer whose address is <ptr>. If the last character read was an end-of-line character, it replaces that character with a null-character, otherwise, it appends a null-character onto the characters read, making a character-string.

If it is successful, meaning it read at least one character, it returns <ptr> as its result, otherwise it returns (char *) NULL as its result and does not alter the target buffer.

SEE ALSO

C Library: fdopen(), fgets(), getc(), puts(), stdin

getutent

Get and decode the next entry in the system's multi-user login file.

SYNOPSIS

```
#include <utmp.h>
struct utmp *getutent();
```

Arguments

None

Returns

The address of the structure containing the information in the record read, or (struct utmp *) NULL) if no record was read

DESCRIPTION

This function reads and decodes the next valid entry in the system's multi-user login file. It saves the information in that record in a static structure (defined below) and it returns the address of that structure as its result. If the function fails to read a record from the system's multi-user login file, the function returns (struct utmp *) NULL) as its result.

If no previous "getutent()" or "getutline()" has been successfully attempted, or "endutent()" has been called since the last call to "getutent()" or "getutline()" this function opens the system's multi-user login file and positions it to the first record in the file. After the function reads a record, it keeps the file open and positions the file to the record immediately following the record read, or to the end of the file if it read no record.

The function "endutent()" closes the system's multi-user login file. Task termination also closes the file. The function "setutent()" rewinds the file so that its current position is the first byte of the first record of the file.

The include-file "<utmp.h>" contains structure and constant definitions used to manipulate data in the system's multi-user login file. The format of the struct utmp structure referenced by the result of this function is as follows:

```
struct utmp
{
char            ut_user[8];
char            ut_id[4];
char            ut_line[12];
unsigned long   ut_time;
};
```

The array "ut_user" contains the login-name of the user, "ut_id" is the entry number (the line number in the file), "ut_line" contains the login device name, and "ut_time" is the system-time when the user logged on. The entries "ut_user" and "ut_line" are only terminated with a null- character if the value contains less than the maximum number of charac- ters. The "ut_id" entry is always four characters.

NOTES

The structure referenced by the result of this function is in static memory and will be overwritten by subsequent calls to "getutent()" and "getutline()".

This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The function will return (struct utmp *) NULL) if the user does not have permission to access the multi-user login file, the current position is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another. The system's multi-user login file is "/atc/utmp".

SEE ALSO

C Library: endutent(), getutline(), setpwent()

Command: who

getutline

Get and decode the next entry in the system's multi-user login file that has specific "ut_line" value.

SYNOPSIS

```
#include <utmp.h>
struct utmp *getutline(line)
char     *line;
```

Arguments

line¦ The address of a character-string containing the"ut_line" value to search for

Returns

The address of the structure containing the information in the record read, or (struct utmp *) NULL if no record was read

DESCRIPTION

This function reads and decodes the next valid entry in the system's multi-user login file which has the login-device name ("ut_line" entry) contained in the character-string referenced by <line>, beginning with the record read by the most recent "getutent()" or "getutline()" call, if any. The function saves the information in a static structure (defined below) and returns the address of that structure as its result. If it couldn't read a record from the system's multi-user login file, or couldn't find a record with a matching login-device name, the function returns (struct utmp *) NULL as its result.

If no previous "getutent()" or "getutline()" has been successfully attempted, or "endutent()" has been called since the last call to "getutent()" or "getutline()" this function opens the system's multi-user login file and positions it to the first record in the file. After the function completes, it leaves the system's multi-user login file open, positioned to the first byte of the next record following the record read, or to the end of the file if none was read.

The function "endutent()" closes the system's multi-user login file. Task termination also closes the file. The function "setutent()" rewinds the file.

The include-file "<utmp.h>" defines structures and constants used to manipulate information in the system's multi-user login file. The format of the struct utmp structure referenced by the result of this function is as follows:

```
struct utmp
{
char            ut_user[8];
char            ut_id[4];
char            ut_line[12];
unsigned long   ut_time;
};
```

The array "ut_user" contains the login-name of the user, "ut_id" is the entry number (the line number in the file), "ut_line" contains the login device name, and "ut_time" is the system time when the user logged on. The entries "ut_user" and "ut_line" are only terminated with a null-character if the value contains less than the maximum number of characters. The "ut_id" entry is always four characters.

NOTES

The structure referenced by the result of this function is in static memory and will be overwritten by subsequent calls to "getutent()" and "getutline()".

This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The function will return (struct utmp *) NULL if the user does not have permission to access the multi-user login file, the current position is end-of-file, or the user has the maximum number of standard I/O streams open and can not open another.

This function begins its search with the most recently read by "getutent()" or "getutline()". To avoid finding the same line again, repeated searches using the same login-device name must first void the "ut_line" entry in the structure referenced by the most recent call to "getutent()" or "getutline()" by setting it to a null-string. The system's multi-user login file is "/act/utmp".

SEE ALSO

C Library: endutent(), getutline(), setpwent()

Command: who

getw

Read a word from a standard I/O stream.

SYNOPSIS

```
int getw(stream)
FILE    *stream;
```

Arguments

stream¦ The standard I/O stream to read from

Returns

The value read if successful, EOF otherwise

DESCRIPTION

This function reads the next sizeof(short) bytes from the stream <stream>, assigns them to a short, casts that short into an int, and returns that value as its result. If the function detects an error or reaches the end of the stream, it returns EOF.

NOTES

The value EOF is a valid value to read, so the functions "ferror()" and "feof()" should be used to check for error and end-of-file conditions on the stream. The function ignores odd bytes at the end of the stream. The function has no boundary alignment requirements.

SEE ALSO

C Library: fdopen(), fopen(), getc(), putw()

gmtime

Break down a system-time value into units in the Greenwich Mean Time zone.

SYNOPSIS

```
#include <time.h>
struct tm *gmtime(pclock)
long     *pclock;
```

Arguments

The address of a system-time value

Returns

The address of the structure describing the system-time value

DESCRIPTION

This function takes the system-time value referenced by the argument <pclock> and breaks it down into the year, month of the year (0-11), day of the month (1-31), day of the week (0-6, Sunday is 0), day of the year (0-365), hour (0-23), minute (0-59), and second (0-59). It saves that information in a structure and returns as its result the address of that structure.

The include-file "<time.h>" defines the structure referenced by the result of this function. That structure is:

```
struct tm
{
int       tm_sec;
int       tm_min;
int       tm_hour;
int       tm_mday;
int       tm_mon;
int       tm_year;
int       tm_wday;
int       tm_yday;
int       tm_isdst;
};
```

The "tm_sec" entry is the number of seconds into the minute and ranges from 0 to 59, "tm_min" is the number of minutes into the hour and ranges from 0 to 59, "tm_hour" is the number of hours into the day and ranges from 0 to 23, "tm_mday" is the day of the month and ranges from 1 to 31, and "tm_mon" is the month of the year and ranges from 0 to 11. The "tm_year" entry is the number of years since 1900, "tm_wday" is the number of days into the week and ranges from 0 to 6, "tm_yday" is the number of days into the year and ranges from 0 to 365, and "tm_isdst" is always zero.

NOTES

The system-time value is expressed in seconds since the epoch. The operating system defines the epoch as 00:00 (midnight) GMT, January 1, 1980.

The structure referenced by the result of this function is in static memory and is modified by subsequent calls to "ctime()", "gmtime()", or "localtime()".

SEE ALSO

C Library: asctime(), ctime(), localtime()

System Call: time()

Command: udate

index

Find the first occurrence of a character in a character-string.

SYNOPSIS

```
char *index(s, c)
char     *s;
char      c;
```

Arguments

s¦ The address of the character-string to search
c¦ The search character

Returns

The address of the first occurrence of the character in the string, or (char *) NULL if the string does not contain the character

DESCRIPTION

This function searches the character-string referenced by <s> for the first occurrence of the character <c>. If the string contains the character, the function returns as its result the address of the first occurrence of the character in the character-string. Otherwise, it returns (char *) NULL.

NOTES

This function is obsolete. It is only included for compatability with older UniFLEX C libraries. New applications should use "strchr()".

SEE ALSO

C Library: rindex(), strchr(), strrchr()

isalnum

Determine if a value is an alphabetic character or a decimal digit.

SYNOPSIS

```
#include <ctype.h>
int isalnum(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is an alphabetic character or a decimal digit, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it an alphabetic character or a decimal digit. Alphabetic characters are the characters ('A'-'Z') and ('a'-'z') inclusive. Decimal digits are the characters ('0'-'9') inclusive. If <c> is an alphabetic character or a decimal digit, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII char- acter or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isalpha

Determine if a value is an alphabetic character.

SYNOPSIS

```
#include <ctype.h>
int isalpha(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is an alphabetic character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it an alphabetic character. Alphabetic characters are the characters ('A'-'Z') and ('a'-'z') inclusive. If <c> is an alphabetic character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isascii

Determine if a value is an ASCII character.

SYNOPSIS

```
#include <ctype.h>
int isascii(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

1 if <c> is a valid ASCII character, 0 otherwise

DESCRIPTION

This function examines the value <c> and determines if it a valid ASCII character. Valid ASCII characters are the values between 0x00 and 0x7F (decimal values 0 through 255) inclusive. If <c> is a valid ASCII character, the function returns 1, otherwise it returns 0.

NOTES

This function is implemented as a macro. However, it will have no side-effects and produces a valid result for all values in the range of an int. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

iscntrl

Determine if a value is a control character.

SYNOPSIS

```
#include <ctype.h>
int iscntrl(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a control character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a control character. Control characters are the values 0x00 through 0x3F inclusive and 0x7F. If <c> is a control character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isdigit

Determine if a value is a decimal digit.

SYNOPSIS

```
#include <ctype.h>
int isdigit(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a decimal digit, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a decimal digit. Decimal digits are the characters ('0'-'9') inclusive. If <c> is a decimal digit, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isgraph

Determine if a value is a graphics character.

SYNOPSIS

```
#include <ctype.h>
int isgraph(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a graphics character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a graphics character. Graphic characters are alphabetic characters, decimal digits, and punctuation characters which are not white-space characters. If <c> is a graphics character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

islower

Determine if a value is a lower-case alphabetic character.

SYNOPSIS

```
#include <ctype.h>
int islower(c)
int  c;
```

Arguments

c| The value to examine

Returns

Non-zero if the value is a lower-case alphabetic character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a lower-case alphabetic character. Lower-case alphabetic characters are the charac- ters ('a'-'z') inclusive. If <c> is a lower-case alphabetic character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isprint

Determine if a value is a printable character.

SYNOPSIS

```
#include <ctype.h>
int isprint(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a printable character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a printable character. Printable characters are alphabetic characters, decimal digits, and punctuation characters. If <c> is a printable character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

ispunct

Determine if a value is a punctuation character.

SYNOPSIS

```
#include <ctype.h>
int ispunct(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a punctuation character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a punctuation character. Punctuation characters are all characters which are not alphabetic characters, decimal digits, white-space characters, or control characters. If <c> is a punctuation character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isspace

Determine if a value is a white-space character.

SYNOPSIS

```
#include <ctype.h>
int isspace(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a white-space character, zero otherwise.

DESCRIPTION

This function examines the value <c> and determines if it a white-space character. White-space characters are the space-character the horizontal-tab character (' t'), the end-of-line character (EOL, ' r', ' n'), and the line-feed character (0x0A). If <c> is a white-space character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type. The C compiler translates the character ' n' to the line-feed character if the "cc" command is called with the "+u" option.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

Command: cc

isupper

Determine if a value is an upper-case alphabetic character.

SYNOPSIS

```
#include <ctype.h>
int isupper(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is an upper-case alphabetic character, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it an upper-case alphabetic character. Upper-case alphabetic characters are the characters ('A'-'Z') inclusive. If <c> is an upper-case alphabetic character, the function returns a non-zero value, otherwise it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII character or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isxdigit(), toascii(), tolower(), _tolower(), toupper(), _toupper()

isxdigit

Determine if a value is a hexadecimal digit.

SYNOPSIS

```
#include <ctype.h>
int isxdigit(c)
int  c;
```

Arguments

c¦ The value to examine

Returns

Non-zero if the value is a hexadecimal digit, zero otherwise

DESCRIPTION

This function examines the value <c> and determines if it a hexadecimal digit. Hexadecimal digits are the characters ('0'-'9'), ('a'-'f'), and ('A'-'F') inclusive. If <c> is a hexadecimal digit, the function returns a non-zero value; otherwise, it returns zero.

NOTES

This function is implemented as a macro. It will have no side-effects but its behavior is unpredictable if <c> is not a valid ASCII char- acter or EOF. The argument <c> will be cast into an int if it is not already of that type.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), toascii(), tolower(), _tolower(), toupper(), _toupper()

_itostr

Convert an int to a character-string.

SYNOPSIS

```
char *_itostr(i, base, digits, psign)
int       i;
int       base;
char     *digits;
int      *psign;
```

Arguments

i|	The value to convert
base|	The base to use while converting
digits|	The digits to use while converting
psign|	The address of a flag to set to indicate the sign of the value or (int NULL if none

Returns

The address of the generated character-string

DESCRIPTION

This function converts the int value <i> to its value represented in the base <base> using the digits in the character-string whose address is <digits>. If <psign> is (int *) NULL, the conversion is an unsigned conversion. Otherwise, the value referenced by <psign> is set to zero if <i> is equal to or greater than zero, non-zero otherwise.

The function returns as its result the address of the character-string it generated, or (char *) NULL if the function detects an error. Possible errors are a <base> less than or equal to one, or not enough digits in the character-string referenced by <digits> for the base.

NOTES

The character-string referenced by the result is in static memory and is overwritten by subsequent calls to this or other conversion functions. The longest character-string this function can generate is 32 charcters.

SEE ALSO

C Library: atoi(), _ltostr()

_l2tos

Convert two-byte integers to short integers.

SYNOPSIS

```
void _l2tos(sp, cp, n)
short    *sp;
char     *cp;
int       n;
```

Arguments

sp¦ The address of the buffer to contain the short integers

cp¦ The address of the buffer containing the two-byte integers

n¦ The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> two-byte integers packed in the array of char referenced by <cp>, saving the converted values in the array of short referenced by <sp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: l3tol(), _l4tol(), ltol3(), _ltol4(), _stol2()

l3tol

Convert three-byte integers to long integers.

SYNOPSIS

```
void l3tol(lp, cp, n)
long     *lp;
char     *cp;
int       n;
```

Arguments

lp| The address of the buffer to contain the long integers
cp| The address of the buffer containing the three-byte integers
n| The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> three-byte integers packed in the array of char referenced by <cp>, saving the converted values in the array of long referenced by <lp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: _l2tos(), _l4tol(), ltol3(), _ltol4(), _stol2()

_l4tol

Convert four-byte integers to long integers.

SYNOPSIS

```
void _l4tol(lp, cp, n)
long     *lp;
char     *cp;
int       n;
```

Arguments

lp¦ The address of the buffer to contain the long integers
cp¦ The address of the buffer containing the four-byte integers
n¦ The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> four-byte integers packed in the array of char referenced by <cp>, saving the converted values in the array of long referenced by <lp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: _l2tos(), l3tol(), ltol3(), _ltol4(), _stol2()

localtime

Break down a system-time value into units in the local time zone.

SYNOPSIS

```
#include <time.h>
struct tm *localtime(pclock)
long     *pclock;
```

Arguments

The address of a system-time value

Returns

The address of the structure describing the system-time value

DESCRIPTION

This function takes the system-time value referenced by the argument <pclock> and breaks it down into the year, month of the year (0-11), day of the month (1-31), day of the week (0-6, Sunday is 0), day of the year (0-365), hour (0-23), minute (0-59), and second (0-59) in the current time zone, applying the standard U. S. A. daylight-savings time conversion if necessary. It saves that information in a structure and returns as its result the address of that structure.

The include-file "<time.h>" defines the structure refered to by the result of this function. That definition is:

```
struct tm
{
int       tm_sec;
int       tm_min;
int       tm_hour;
int       tm_mday;
int       tm_mon;
int       tm_year;
int       tm_wday;
int       tm_yday;
int       tm_isdst;
};
```

The "tm_sec" entry is the number of seconds into the minute and ranges from 0 to 59, "tm_min" is the number of minutes into the hour and ranges from 0 to 59, "tm_hour" is the number of hours into the day and ranges from 0 to 23, "tm_mday" is the day of the month and ranges from 1 to 31, and "tm_mon" is the month of the year and ranges from 0 to 11. The "tm_year" entry is the number of years since 1900, "tm_wday" is the number of days into the week and ranges from 0 to 6, "tm_yday" is the number of days into the year and ranges from 0 to 365, "tm_isdst" is one if the standard U. S. A. daylight-savings time conversion was applied, zero otherwise.

NOTES

The system time value is expressed in seconds since the epoch. The operating system defines the epoch as 00:00 (midnight) GMT, January 1, 1980. The structure referenced by the result of this function is in static memory and is modified by subsequent calls to "ctime()", "gmtime()", or "localtime()".

The function applies the standard U. S. A. daylight-savings time conversion only if the current system configuration indicates that daylight-savings time is in effect. If standard U. S. A. daylight-savings time is in effect, the function adds an hour to the time if the time falls between 02:00 AM on the last Sunday in April and 01:00 AM on the last Sunday in October.

This function calls "tzset()" if necessary, setting the global vari- ables "daylight", "timezone", and "tzname".

SEE ALSO

C Library: asctime(), ctime(), daylight, gmtime(), timezone, tzname, tzset()

System Call: time()

Command: date

longjmp

Perform a non-local goto.

SYNOPSIS

```
#include <setjmp.h>
void longjmp(env, val)
jmp_buf   env;
int       val;
```

Arguments

env¦ Contains environmental information about the target of the non-local goto

val¦ The value to return as the apparent result of the "setjmp()" associated with <env>

Returns

Never

DESCRIPTION

This function restores the program execution environment to that described by the argument <env>. The effect is that of a goto to the "setjmp()" call which saved the environmental information in the argument <env>; with <arg>, if <arg> is not zero, or 1; otherwise, as the apparent result of the "setjmp()" call.

NOTES

Statements following the call to this function will never be executed. The scope containing the "setjmp()" call which set up the <env> argument must not have executed a return or the result of this function is unpredictable. All variables allocated to a register are restored to their value at the "setjmp()" call.

SEE ALSO

C Library: setjmp()

ltol3

Convert long integers to three-byte integers.

SYNOPSIS

```
void ltol3(cp, lp, n)
char     *cp;
long     *lp;
int       n;
```

Arguments

cp	The address of the buffer to contain the three-byte integers
lp	The address of the buffer containing the long integers
n	The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> long integers in the array referenced by <lp> to three-byte integers, saving the converted values packed into the array of char referenced by <cp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: _l2tos(), l3tol(), _l4tol(), _ltol4(), stol2()

_lto14

Convert long integers to four-byte integers.

SYNOPSIS

```
void _lto14(cp, lp, n)
char     *cp;
long     *lp;
int       n;
```

Arguments

cp¦	The address of the buffer to contain the four-byte integers
lp¦	The address of the buffer containing the long integers
n¦	The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> long integers in the array referenced by <lp> to four-byte integers, saving the converted values packed into the array of char referenced by <cp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: _sto12(), _l2tos(), l3tol(), _l4tol(), ltol3(), stol2()

_ltostr

Convert a long to a character-string.

SYNOPSIS

```
char *_ltostr(i, base, digits, psign)
long      i;
int       base;
char     *digits;
int      *psign;
```

Arguments

i¦	The value to convert
base¦	The base to use while converting
digits¦	The digits to use while converting
psign¦	The address of a flag to set to indicate the sign or (int *) NULL

Returns

The address of the generated character-string

DESCRIPTION

This function converts the long value <i> to its value represented in the base <base> using the digits in the character-string referenced by <digits>. If <psign> is (int *) NULL, the conversion is an unsigned conversion. Otherwise, the value referenced by <psign> is set to zero if <i> is equal to or greater than zero, non-zero otherwise.

The function returns as its result the address of the character-string it generated, or (char *) NULL if the function detected an error. Possible errors are a <base> less than or equal to one, or not enough digits in the character-string referenced by <digits> for the base <base>.

NOTES

The character-string referenced by the result is in static memory and is overwritten by subsequent calls to this or other conversion functions. The longest character-string this function can generate is 32 charcters.

SEE ALSO

C Library: atol(), _itostr()

malloc

Allocate memory.

SYNOPSIS

```
char *malloc(nbytes)
unsigned  nbytes;
```

Arguments

nbytes The number of bytes to allocate

Returns

The address of the allocated block of memory or (char *) NULL if none was available

DESCRIPTION

This function allocates <nbytes> bytes of memory from the arena of available memory. It returns the address of the first byte of the allocated memory or (char *) NULL if none was available.

The first byte of the allocated memory is properly aligned for any use.

NOTES

The function "free()" returns allocated memory to the arena of available memory.

SEE ALSO

C Library: calloc(), free(), realloc()

System Call: brk(), cdata(), sbrk()

memccpy

Copy memory.

SYNOPSIS

```
#include <memory.h>
char *memccpy(ptr1, ptr2, c, n)
char     *ptr1;
char     *ptr2;
int       c;
int       n;
```

Arguments

ptr1	The target buffer address
ptr2	The source buffer address
c	The stop-value
n	The maximum number of bytes to copy

Returns

The address of the byte following the copy of the stop-value <c> in the target buffer, or (char *) NULL if <c> was not found

DESCRIPTION

This function copies bytes from the buffer whose address is <ptr2> to the buffer whose address is <ptr1> until either the value <c> is copied or the requested number of bytes has been copied, whichever comes first. It returns as its result the address of the byte following the copy of the value <c> in the target buffer, or (char *) NULL if that value was not found.

NOTES

The behavior of overlapping copy operations is not defined and may behave differently on different systems. If <n> is less than or equal to zero, the function copies no data and returns (char *) NULL as its result. The include-file "<memory.h>" defines this and other block memory functions.

SEE ALSO

C Library: memchr(), memcmp(), memcpy(), memset()

memchr

Find a value in a block of memory.

SYNOPSIS

```
#include <memory.h>
char *memchr(ptr, c, n)
char     *ptr;
int       c;
int       n;
```

Arguments

ptr	The address of the buffer to search
c	The value to search for
n	The maximum number of bytes to search

Returns

The address of the first byte with the value <c>, or (char *) NULL if <c> was not found

DESCRIPTION

This function searches the first <n> bytes in the buffer whose address is <ptr> for the value <c>. If the value is found, it returns the address of that value as its result. Otherwise, it returns (char *) NULL.

NOTES

If <n> is less than or equal to zero, the function always returns (char *) NULL. The include-file "<memory.h>" defines this and other block memory functions.

SEE ALSO

C Library: memccpy(), memcmp(), memcpy(), memset()

memcmp

Compare two blocks of memory.

SYNOPSIS

```
#include <memory.h>
int memcmp(ptr1, ptr2, n)
char     *ptr1;
char     *ptr2;
int       n;
```

Arguments

ptr1	The address of the first buffer to compare
ptr2	The address of the second buffer to compare
n	The maximum number of bytes to compare

Returns

A value less than, equal to, or greater than zero, if the buffer referenced by <ptr1> is lexicographically less than, equal to, or greater than the buffer referenced by <ptr2>

DESCRIPTION

This function lexicographically compares the buffer referenced by <ptr1> with the buffer referenced by <ptr2> and returns as its result a value which indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the buffer referenced by <ptr1> is lexicographically less than, equal to, or greater than the buffer referenced by <ptr2>.

NOTES

If <n> is less than or equal to zero, the result of this function is always zero. A non-zero result is the result of subtracting the differing character in the buffer referenced by <ptr1> from the differing character in the buffer referenced by <ptr2>. The include-file "<memory.h>" defines this and other block memory functions.

SEE ALSO

C Library: memccpy(), memchr(), memcpy(), memset()

memcpy

Copy memory.

SYNOPSIS

```
#include <memory.h>
char *memcpy(ptr1, ptr2, n)
char     *ptr1;
char     *ptr2;
int       n;
```

Arguments

ptr1	The target buffer address
ptr2	The source buffer address
n	The number of bytes to copy

Returns

<ptr1>

DESCRIPTION

This function copies bytes from the buffer whose address is <ptr2> to the buffer whose address is <ptr1> until the requested number of bytes has been copied. It returns as its result the address of the target buffer <ptr1>.

NOTES

The behavior of overlapping copy operations is not defined and may behave differently on different systems. If <n> is less than or equal to zero, the function copies no data. The include-file "<memory.h>" defines this and other block memory functions.

SEE ALSO

C Library: memccpy(), memchr(), memcmp(), memset()

memset

Set a block of memory.

SYNOPSIS

```
#include <memory.h>
char *memset(ptr, c, n)
char     *ptr;
int       c;
int       n;
```

Arguments

```
ptr¦ The address of the buffer to set
c¦    The value to set
n¦    The number of bytes to set
```

Returns

<ptr>

DESCRIPTION

This function sets <n> bytes of memory beginning at the address <ptr> to the value <c>. It returns as its result its argument <ptr>.

NOTES

If <n> is less than or equal to zero, the function modifies no memory. The include-file "<memory.h>" defines this and other block memory functions.

SEE ALSO

C Library: memccpy(), memchr(), memcmp(), memcpy()

mktemp

Generate a unique pathname from a template.

SYNOPSIS

```
char *mktemp(template)
char     *template;
```

Arguments

template| The address of the character-string containing the template for the temporary pathname

Returns

Its argument <template> if it successfully generated a unique pathname, (char *) NULL otherwise

DESCRIPTION

This function generates a unique pathname from the template pathname in the character-string referenced by <template>. A unique pathname is one which does not reach a file but which contains a path that can be followed. It returns <template> as its result if successfully generated a pathname for a file that does not exist, or (char *) NULL otherwise.

If the template pathname ends in six 'x' or 'X' characters, it replaces those characters with an 'A' followed by the five-character representation of the current process-ID. It then checks the filesystem for that pathname. If that pathname doesn't exist, it returns its argument <template>. If that pathname already exists, it changes the 'A' to a 'B' and retrys, continuing until it generates a pathname that does not reference an existing file or it exhausts the upper- and lower-case alphabet.

If the template pathname does not end in 'x' or 'X' characters, the function returns <template> if it is a unique pathname, or (char *) NULL if it is not.

NOTES

If the path in the template pathname can not be followed, or contains a file that is not a directory, the function returns (char *) NULL. If the function returns (char *) NULL, the variable "errno" contains the system error code describing the error.

SEE ALSO

System Call: getpid(), read()

getc

Fetch a character from a stream.

SYNOPSIS

```
#include <stdio.h>
int getc(stream)
FILE     *stream;
```

Arguments

stream| The standard I/O stream from which a character is to be fetched

Returns

The character read if successful, otherwise EOF.

DESCRIPTION

This function reads the next character from the standard I/O stream opened for input referenced by <stream>. It returns that character (as an int) if it was successful, otherwise it returns EOF.

NOTES

The character read is considered to be an unsigned char so there is no sign extension when converting the character to an integer value for returning.

SEE ALSO

C Library: fgetc() getchar() putc()

strtol

Convert a character-string to a long integer.

SYNOPSIS

```
long strtol(str, ptr, base)
char     *str;
char    **ptr;
int       base;
```

Arguments

str| The address of the character-string to convert to a long integer

ptr| The address of the char * to contain the address of the character which terminates the conversion, or (char**) NULL if no assignment is to be made

base| Indicates the base of the digits

Returns

The value generated from the character-string

DESCRIPTION

This function converts the character-string pointed to by <str> to a long integer. It converts using the base specified by <base> and assigns the address of the character ending the conversion to the char * referenced by <ptr>. The character that ends the conversion is either the null-character terminating the string or the first character that was inconsistent with the base. If <ptr> is (char **) NULL, no assignment is made.

if the base <base> is greater than 0 and less than or equal to 36, it describes the base of the digits in the character-string. (For bases between 11 and 36, the alphabetic characters, in order, are used as di- gits.) If the base is 0, the function examines the character-string to determine the base. If following the optional white-space and sign is "0x" or "0X", the base is assumed to be 16. Otherwise, if a '0' follows the optional white-space and sign, the base is assumed to be 8. Otherwise, the base is assumed to be 10. If the base is less than 0 or greater than 36, the base is assumed to be 10.

NOTES

Overflow conditions are ignored.

SEE ALSO

C Library: _atoh() atoi() atol() _atoo() _atos() _strtoi()

printf

Write formatted data to "stdout".

SYNOPSIS

```
#include <stdio.h>
int printf(format [,arglist])
char     *format;
```

Arguments

format¦ The address of a character-string containing a format description

Returns

The number of characters written to "stdout" or EOF if an er- ror occurred

DESCRIPTION

This function generates characters from the format description in the character-string referenced by <format> and the arguments in the argument-list <arglist>, if any, and writes these characters to the standard I/O output stream "stdout". It returns as its result the number of characters written to "stdout".

The format description in the character-string referenced by <format> contains literal characters and field descriptions. The function writes literal characters to "stdout" with no interpretation. The function interprets field descriptions to determine what characters it generates, what type of argument it consumes, if any, from the argument list <arglist>, and the type of conversion it performs. The number of arguments and the type of the arguments in the argument list <arglist> depends on the format description. The argument list can be omitted.

For a complete description of the <format> argument, see the manual page for "fprintf()".

NOTES

The function writes characters to "stdout" using "fputc()". If "stdout" is buffered, standard I/O does not write characters to the file attached to the stream until it fills the stream's buffer or closes the stream. If "stdout" is line-buffered (buffered and attached to a file that is a terminal), standard I/O does not write characters to the file attached to the stream until it fills the stream's buffer, closes the stream, writes an end-of-line character (EOL) to the stream, or reads data from a terminal. The include-file "<stdio.h>" defines the functions and constants available in standard I/O. This file must be included in the C source before the first reference to this function.

SEE ALSO

C Library: ecvt(), fcvt(), fdopen(), fopen(), fprintf(), fputc(), fscanf(), gcvt(), scanf(), sprintf(), sscanf(), stdout

putc

Write a character to a stream.

SYNOPSIS

```
#include <stdio.h>
int putc(c, stream)
char      c;
FILE     *stream;
```

Arguments

c	The character to write
stream	The standard I/O stream to write to

Returns

The value written if successful, EOF otherwise.

DESCRIPTION

This function writes the character <c> to the standard I/O stream <stream>. The function returns the character written as its result if it successfully writes the character to the stream, otherwise, it returns EOF.

NOTES

If the stream is buffered, standard I/O flushes the buffered data whenever the buffer fills or the stream closes. If the stream is line-buffered (buffered and attached to a character-special device), standard I/O flushes the buffered data whenever the buffer fills, the stream closes, a standard I/O function writes an EOL character to the stream, or a standard I/O function reads data from a character-special device (a terminal).

SEE ALSO

C Library: fdopen(), fopen(), fputc(), getc(), putchar()

putchar

Write a character to "stdout".

SYNOPSIS

```
#include <stdio.h>
int putchar(c)
char      c;
```

Arguments

c¦ The character to write

Returns

The value written if successful, EOF otherwise.

DESCRIPTION

This function writes the character <c> to the standard I/O standard output stream "stdout". The function returns the character written as its result if it successfully wrote the character to "stdout", otherwise, it returns EOF.

NOTES

If the stream is buffered, standard I/O flushes the buffered data whenever the buffer fills or the stream closes. If the stream is line-buffered (buffered and attached to a character-special device), standard I/O flushes the buffered data whenever the buffer fills, the stream closes, a standard I/O function writes an EOL character to the stream, or a standard I/O function reads data from a character-special device (a terminal).

SEE ALSO

C Library: fdopen(), fopen(), fputc(), getchar(), putc(), stdout

putpwent

Format and write a system password-file record.

SYNOPSIS

```
#include <pwd.h>
int putpwent(ptr, stream)
struct passwd *ptr;
FILE          *stream;
```

Arguments

ptr¦	Address of a structure containing the information to write
stream¦	The standard I/O stream to write to

Returns

Zero if the record was successfully written, EOF otherwise.

DESCRIPTION

This function generates a character-string from the information in the structure referenced by <ptr> and writes that character-string to the standard I/O output stream <stream>. It generates the character-string in the format required by the system-password file. The function returns zero as its result if it successfully formats and writes the record, otherwise it returns EOF.

The function generates the character-string using the following "sprintf()" format-string:

```
"%s:%s:%d:%s:%s n"
```

The format of the structure referenced by the result of this function is defined by the include-file "<pwd.h>" and is defined as follows:

```
struct passwd
{
char     *pw_name;
char     *pw_passwd;
int       pw_uid;
char     *pw_dir;
char     *pw_shell;
};
```

The entry "pw_name" is the address of a character-string containing the user-name, "pw_passwd" is the address of a character-string containing the encrypted password, "pw_uid" contains the user's identifying number (user-ID), "pw_dir" is the address of a character-string containing the user's initial home-directory, and "pw_shell" is the address of a character-string containing containing the shell-command for the first program to run after logging on.

A null-string as the encrypted password indicates that the user has no password, and a null-string as the shell-command that indicates the initial program is the standard shell.

NOTES

This function uses standard I/O and will enlarge more than expected a program not otherwise using standard I/O. The "pw_passwd" and "pw_shell" entries in the structure referenced by <ptr> may be (char *) NULL. If so, the function uses a null-string when generating the system password record.

SEE ALSO

C Library: endpwent(), getpw(), getpwent(), getpwnam(), getpwuid(), setpwent()

Command: password

puts

Write a character-string to "stdout".

SYNOPSIS

```
#include <stdio.h>
int puts(s)
char     *s;
```

Arguments

s¦ The address of the character-string to write

Returns

Zero if successful, EOF otherwise.

DESCRIPTION

This function writes the character-string referenced by <s>, followed by an end-of-line character (EOL), to the standard I/O standard output stream "stdout". The function returns zero as its result if it successfully writes the character-string to "stdout", otherwise, it returns EOF.

NOTES

The function does not write the null-character terminating the character-string to "stdout".

SEE ALSO

C Library: fdopen(), fopen(), fputs(), gets(), putchar(), stdout

putw

Write a word to a stream.

SYNOPSIS

```
int putw(wd, stream)
int       wd;
FILE     *stream;
```

Arguments

wd¦	The value to write
stream¦	The standard I/O stream

Returns

The value written if successful, EOF otherwise

DESCRIPTION

This function casts the int argument <wd> into a short then writes that short to the standard I/O output stream referenced by <stream>. It writes the high-order byte first, then the low-order byte. It then casts the short written into an int and returns that value as its result. If it detects an error, it returns EOF as its result.

NOTES

The value EOF is a valid value to write, so the functions "ferror()" and "feof()" should be used to check for error and end-of-file conditions for the stream. There are no boundary alignment requirements for writing a word to the stream.

SEE ALSO

C Library: fdopen(), fopen(), getw(), putc()

qsort

Sort data.

SYNOPSIS

```
int qsort(base, nel, width, compar)
char          *base;
unsigned int   nel;
unsigned int   width;
int          (*compar)();
```

Arguments

base¦	The address of the data to sort
nel¦	The number of records in the data
width¦	The number of bytes in a record of data
compar¦	The address of a function to use to compare two records

Returns

0 if it successfully sorted the data, -1 otherwise

DESCRIPTION

This function sorts in place the data at the address <base>. The data contain <nel> records with each record <width> in length. It uses the function whose address is <compar> to compare two records of data. The function returns 0 as its result if it successfully sorted the data, otherwise it returns -1 as its result.

The function whose address is <compar> is a function returning an int with two arguments, both addresses of records of data. The function returns a value less than, equal to, or greater than zero if the record referenced by the first argument is less than, equal to, or greater than the record referenced by the second argument.

NOTES

The only possible condition resulting in a result of -1 is a record size <width> larger than the maximum. The maximum is currently 256 bytes. The argument <base> should be a pointer-to-element cast into a (char *).

SEE ALSO

Command: sort

rand

Generate a random number.

SYNOPSIS

```
int rand();
```

Arguments

None

Returns

A random number

DESCRIPTION

This function generates a random number and returns that value as its result. The number is between 0 and 32767 inclusive.

This function is a pseudo-random number generator, generating the next number in a sequence. The previous number in the sequence is the seed set automatically at the start of the program, the seed set explicitly by the "rrand()" or "srand()" functions, or the previously generated random number.

NOTES

The sequence of numbers generated by this function from a particular seed is always reproducable.

SEE ALSO

C Library: rrand(), srand()

realloc

Reallocate an allocated block of data.

SYNOPSIS

```
char *realloc(buf, size)
char          *buf;
unsigned int   size;
```

Arguments

buf¦	The address of the allocated buffer to reallocate
size¦	The requested new size of the buffer, in bytes

Returns

The address of the reallocated buffer

DESCRIPTION

This function changes the size of the allocated buffer whose address is <buf> to the <size> bytes. If the space allocated to the buffer is large enough to accomodate <size> bytes, the function returns from the buffer as much space as possible to the arena of available memory and returns <buf> as its result. Otherwise, it returns the buffer <buf> to the arena of available memory and allocates a buffer of <size> bytes. If successful, it copies the data in the original buffer to the newly allocated buffer and returns the address of the allocated buffer as its result. Otherwise, it returns (char *) NULL as its result.

NOTES

The original block is destroyed if the function returns (char *) NULL.

SEE ALSO

C Library: calloc(), free(), malloc()

System Call: brk(), cdata(), sbrk()

rewind

Rewind a stream.

SYNOPSIS

```
#include <stdio.h>
int rewind(stream)
FILE     *stream;
```

Arguments

stream¦ The standard I/O stream to rewind

Returns

Void

DESCRIPTION

This function rewinds the stream referenced by <stream>. Rewinding a stream positions the stream to the beginning of its attached file.

NOTES

This function undoes the effect of an "ungetc()" function. This function does not rewind a stream that is attached to a character-special file (terminal).

SEE ALSO

C Library: fdopen(), fopen(), fseek(), ftell()

System Call: lseek()

rindex

Find the last occurrence of a character in a character-string.

SYNOPSIS

```
char *rindex(s, c)
char     *s;
char      c;
```

Arguments

s¦ The address of the character-string to search
c¦ The character to search for

Returns

The address of the last occurrence of the character in the string, or (char *) NULL if the string does not contain the character

DESCRIPTION

This function searches the character-string whose address is <s> for the last occurrence of the character <c>. If the string contains the character, the function returns as its result the address of the last occurrence of the character. Otherwise, it returns (char *) NULL.

NOTES

This function is obsolete. It is only included for compatability with older UniFLEX C libraries. New applications should use "strrchr()".

SEE ALSO

C Library: index(), strchr(), strrchr()

rrand

Set the seed of the random number generator to a value generated from the current system-time value.

SYNOPSIS

```
void rrand(seed);
```

Arguments

None

Returns

Void

DESCRIPTION

This function sets the seed of the pseudo-random number generator to a value generated from current system-time value. The value generated is that of the low 15 bits of the system-time value.

NOTES

The system-time value is the current time expressed in the number of seconds since the epoch. The system defines the epoch as 00:00 (midnight) on January 1, 1980, Greenwich Mean Time. The seed is the value from which the next random number is generated. The random number generating function "rand()" always generates the same sequence of random numbers from a particular seed.

SEE ALSO

C Library: rand(), srand()

System Call: time()

scanf

Read and interpret formatted data from "stdin".

SYNOPSIS

```
#include <stdio.h>
int scanf(format [, addrlist])
char     *format;
```

Arguments

format¦ The address of a character-string containing a format description

Returns

The number of items in the address-list <addrlist> that it successfully assigns or EOF if an error occurs before it assigns any data

DESCRIPTION

This function reads and interprets data from the standard I/O stream "stdin", according to the format description in the character-string referenced by <format>. Following the argument <format> in the argument list, the function expects a list of address of variables to receive the values it generates from the data it reads from "stdin", if any. The function returns as its result the number of assignments is makes, or EOF if it encounters an error before making the first assignment.

The argument <format> is a character-string containing a format description, which describes the format of the data read from "stdin". The format description consists of literal characters, white-space characters, and field descriptions, in any sequence.

Literal characters are all characters which are not white-space characters (as defined by "isspace()"), and not part of field descriptions. A literal character tells the function to match that character with the next character read from "stdin". If it does not match exactly, the function ends.

White-space characters are the space (' '), end-of-line (' n'), horizontal-tab (' t'), form-feed (' f'), and carriage-return (' r) characters. A white-space character tells the function to read and consume characters from "stdin" until it reaches a character which is not a white-space character or it reaches the

end of the data. The next character available from "stdin" is the next character which is not a white-space character. The function does nothing with a white-space character in the format description if the next character from "stdin" is not a white-space character.

A field description tells the function how to interpret the next character or characters read from "stdin". It tells the function the maximum number of characters to read, the form of the characters read, the type of value to assign any result to, and whether to perform an assignment. A field description has the following syntax:

%[*][<width>][<flags>]<type>

Where

- The '%' character introduces the field description.
- The '*' character tells the function to suppress assigning the interpreted value to a variable.
- The <width> part tells the function the maximum number of characters to read to satisfy the field (including leading white-space characters, if the field type skips leading white-space characters).
- The <flags> part alters the type of assignment made by the function, and may be either the 'h' or the 'l' character.
- The <type> part defines the type of the field and may be any one of the characters in the following string:

  "cdefgosux%[".

For a complete description of the function's field types, see the manual pages for the standard I/O function "fscanf()".

NOTES

The most common mistake made when using this function is passing to the function the values of the variables to receive the results of this function, instead of the addresses of those variables. The include-file "<stdio.h>" defines this function, other functions, macros, and constants used by standard I/O.

SEE ALSO

C Library: fdopen(), fopen(), fprintf(), fscanf(), printf(), sprintf(), sscanf(), stdin

setbuf

Set buffering attributes of a stream.

SYNOPSIS

```
#include <stdio.h>
void setbuf(stream, buf)
FILE     *stream;
char     *buf;
```

Arguments

stream¦ The standard I/O stream whose buffer characteristics are being set

buf¦ The address of the buffer to use as the stream's buffer, or (char *) NULL if the stream is to be unbuffered

Returns

Void

DESCRIPTION

This function sets the buffering characteristics for the standard I/O stream referenced by <stream>. If <buf> is (char *) NULL, the stream is set for unbuffered I/O. Otherwise, the stream is set for buffered I/O with <buf> set as address of the buffer to use for the buffered I/O.

NOTES

The buffer whose address is <buf> is assumed to contain at least BIG- BUF bytes. The include-file <stdio.h> contains this and other definitions for standard I/O. This function should only be used before any I/O is performed on the stream. If I/O has been performed on the stream, the current buffering will be lost.

SEE ALSO

C Library: fdopen(), fopen()

setjmp

Setup for a non-local goto.

SYNOPSIS

```
#include <setjmp.h>
int setjmp(env)
jmp_buf   env;
```

Arguments

env| The value to receive the current environmental information

Returns

Zero when returning from "setjmp()", non-zero when the result of a "longjmp()"

DESCRIPTION

This function saves the current environmental information in the argument <env> so that a subsequent call to "longjmp()" with <env> as its argument will result with execution continuing as though the "setjmp()" call had returned. The effect of a "longjmp()" using <env> as its argument is that of a goto from the "longjmp()" call to the "setjmp()" call.

A 0 result indicates that "setjmp()" is returning after setting the argument <env> with the current environmental information. A non-zero result indicates that "longjmp()" was called with an argument <env>.

NOTES

The scope calling this function must not have returned by the time "longjmp()" is called with the argument <env> or the result of the "longjmp()" call will be unpredictable. Values residing in registers (those defined as register variables that have had registers assigned to them) will revert to their value at the time of the "setjmp()" call when "longjmp()" is called with the argument <env>.

The argument <env> is actually the address of a structure for the current environmental information. The include-file <setjmp.h> contains the typedef for "jmp_buf" and other information used by "setjmp()" and "longjmp()".

SEE ALSO

C Library: longjmp()

setpwent

Reset password-file handling.

SYNOPSIS

```
#include <pwd.h>
void setpwent();
```

Arguments

None

Returns

Void

DESCRIPTION

This function resets password-file handling initiated by "getpwent()", "getpwnam()", or "getpwuid()". It reinitializes the resources allocated by, and rewinds the files opened by, those routines.

NOTES

This function does nothing if "getpwent()", "getpwnam()", or "getpwuid()" has not been called or "endpwent()" has been called since the last call to one of those routines.

SEE ALSO

C Library: endpwent(), getpwent(), getpwnam(), getpwuid()

setutent

Reset multi-user login-file handling.

SYNOPSIS

```
#include <utmp.h>
void setutent();
```

Arguments

None

Returns

Void

DESCRIPTION

This function resets multi-user login-file handling initiated by "getutent()" or "getutline()". It reinitializes the resources allocated to and rewinds the files opened by those routines.

NOTES

This function does nothing if neither "getutent()" nor "getutline()" has been called or "endutent()" has been called since the last call to either of those routines.

SEE ALSO

C Library: endutent(), getutent(), getutline()

sleep

Suspend execution for an interval.

SYNOPSIS

```
unsigned int sleep(time)
unsigned int   time;
```

Arguments

time¦ The maximum number of seconds to suspend execution

Returns

The number of seconds remaining in the requested interval

DESCRIPTION

This function requests that the execution of the current task be suspended for the number of seconds specified by the argument <time>. It returns after the requested interval has passed or an alarm-, hangup-, keyboard-, or quit-interrupt has been caught. It returns as its result the number of seconds remaining in the requested sleep interval.

This function is implemented using the "alarm" system call. It requests that an alarm-interrupt be sent to the current task in <time> seconds, then pauses using the "pause()" function, waiting for a signal. The function knows about an alarm-interrupt request armed before the function is called.

If the armed alarm-interrupt request is scheduled to take place during the sleep interval, the function pauses for time remaining on the armed alarm-interrupt request. Then, if that interval passes completely, it resignals the alarm-interrupt so the user can handle it. Otherwise, or if the armed alarm-interrupt request is scheduled to take place after the sleep interval is complete, upon return from the "pause()" function, the function rearms the alarm-interrupt for the time remaining, and restores the signalling information for the alarm- interrupt to the state before the function was called.

NOTES

Requesting a sleep interval <time> of 0 results in the task pausing until the next signal.

SEE ALSO

System Call: alarm(), signal(), wait()
Command: sleep

sprintf

Generate a character-string containing formatted data.

SYNOPSIS

```
#include <stdio.h>
int sprintf(string, format [,arglist])
char     *string;
char     *format;
```

Arguments

string¦	The address of a buffer to contain the generated string
format¦	The address of a character-string containing a format description

Returns

The number of characters written to "stdout" or EOF if an er- ror occurred

DESCRIPTION

This function generates characters from the format description in the character-string referenced by <format> and the arguments in the argument-list <arglist>, if any, writes these characters into the buffer whose address is <string>, then appends a null-character onto those generated characters in that buffer. It returns as its result the length of the generated character-string.

The format description in the character-string referenced by <format> contains literal characters and field descriptions. The function copies literal characters to character-string with no interpretation. The function interprets field descriptions to determine what characters it generates, what type of argument it consumes, if any, from the argument list <arglist>, and the type of conversion it performs. The number of arguments and the type of the arguments in the argument list <arglist> depends on the format description. The argument list can be omitted.

For a complete description of the <format> argument, see the manual page for "fprintf()".

NOTES

The function assumes that the buffer whose address is <string> is large enough to hold the character-string it generates. The include-file "<stdio.h>" defines this function and other functions and constants available in standard I/O. This file must be included in the C source before the first reference to this function.

SEE ALSO

C Library: ecvt(), fcvt(), fdopen(), fopen(), fputc(), fscanf(), gcvt(), printf(), scanf(), sprintf(), sscanf(), stdout

srand

Set the seed of the random number generator.

SYNOPSIS

```
void srand(seed)
int       seed;
```

Arguments

seed| The seed for the random number generator

Returns

Void

DESCRIPTION

This function sets the seed of the pseudo-random number generator to a value generated from the argument <seed>. The value generated is that of the low 15 bits of the argument.

NOTES

The seed is the value from which the next random number is generated. The random number generating function "rand()" always generates the same sequence of random numbers from a particular seed.

SEE ALSO

C Library: rand(), rrand()

sscanf

Interpret formatted data from a character-string.

SYNOPSIS

```
#include <stdio.h>
int sscanf(string, format [, addrlist])
char     *string;
char     *format;
```

Arguments

string¦	The string containing data to interpret
format¦	The address of a character-string containing a format description

Returns

The number of items in the address-list <addrlist> that it successfully assigns or EOF if an error occurs before it assigns any data

DESCRIPTION

This function interprets data from the character-string referenced by the argument <string>, according to the format description in the character-string referenced by <format>. Following the argument <format> in the argument list, the function expects a list of address of variables to receive the values it generates from the characters in the data character-string, if any. The function returns as its result the number of assignments is makes, or EOF if it encounters an error before making the first assignment.

The argument <format> is a character-string containing a format descrip- tion, which describes the format of the characters in the data character-string. The format description consists of literal characters, white-space characters, and field descriptions, in any sequence.

Literal characters are all characters which are not white-space charac- ters (as defined by "isspace()"), and not part of field descriptions. A literal character tells the function to match that character with the next character in the data character-string. If it does not match exactly, the function ends.

White-space characters are the space (' '), end-of-line (' n'), horizontal-tab (' t'), form-feed (' f'), and carriage-return (' r) characters. A white-space character tells the function to skip characters in the data character-string until it reaches a character which is not a white-space character or it reaches the end of the data. The next character available from the data character-string is the next character which is not a white-space character. The function does nothing with a white-space character in the format description if the next character from the data character-string not a white-space character.

A field description tells the function how to interpret the next character or characters from the data character-string. It tells the functior the maximum number of characters to get, the form of those characters, the type of value to assign any result to, and whether to perform an assignment. A field description has the following syntax:

%[*][<width>][<flags>]<type>

The '%' character introduces the field description. The '*' character tells the function to suppress assigning the interpreted value to a variable. The <width> part tells the function the maximum number of characters to get to satisfy the field (including leading white-space characters, if the field type skips leading white-space characters). The <flags> part alters the type of assignment made by the function, and may be either the 'h' or the 'l' character. The <type> part defines the type of the field and may be any one of the characters in the following string: "cdefgosux%[".

For a complete description of the function's field types, see the manual pages for the standard I/O function "fscanf()".

NOTES

The most common mistake made when using this function is passing to the function the values of the variables to receive the results of this function, instead of the addresses of those variables. The include-file "<stdio.h>" defines this function, other functions, macros, and constants used by standard I/O.

SEE ALSO

C Library: fdopen(), fopen(), fprintf(), fscanf(), printf(), scanf(), sprintf(), stdin

_stol2

Convert short integers to two-byte integers.

SYNOPSIS

```
void _stol2(cp, sp, n)
char     *cp;
short    *sp;
int       n;
```

Arguments

cp| The address of the buffer to contain the two-byte integers

sp| The address of the buffer containing the short integers

n| The number of values to convert

Returns

Void

DESCRIPTION

This function converts <n> short integers in the array referenced by <sp> to two-byte integers, saving the converted values packed into the array of char referenced by <cp>. The function returns no result.

NOTES

This function is typically used to avoid addressing problems resulting from misaligned addresses.

SEE ALSO

C Library: _l2tos(), l3tol(), _l4tol(), ltol3(), _ltol4()

strcat

Concatenate one character-string onto another.

SYNOPSIS

```
#include <string.h>
char *strcat(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1	The address of the target character-string
s2	The address of the character-string to concatenate onto <s1>

Returns

<s1>

DESCRIPTION

This function appends a copy of the character-string referenced by <s2> onto the character-string referenced by <s1>. It returns <s1> as its result.

NOTES

The resulting character-string is always terminated with a null-character. The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strncat()

strchr

Find the first occurrence of a character in a character-string.

SYNOPSIS

```
#include <string.h>
char *strchr(s, c)
char     *s;
char      c;
```

Arguments

s	The address of the character-string to search
c	The character to search for

Returns

The address of the first occurrence of the character in the string, or (char *) NULL if the string does not contain the character

DESCRIPTION

This function searches the character-string whose address is <s> for the first occurrence of the character <c>. If the string contains the character, the function returns as its result the address of the first occurrence of the character. Otherwise, it returns (char *) NULL.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: index(), strrchr()

strcmp

Compare two character-strings.

SYNOPSIS

```
#include <string.h>
int strcmp(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1	The address of the first string to compare
s2	The address of the second string to compare

Returns

A value less than, equal to, or greater than zero, if the character-string referenced by <s1> is lexicographically less than, equal to, or greater than the character-string referenced by <s2>

DESCRIPTION

This function lexicographically compares the character-string referenced by <s1> with the character-string referenced by <s2> and returns as its result a value which indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the character-string referenced by <s1> is lexicographically less than, equal to, or greater than the character-string referenced by <s2>.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strncmp()

strcpy

Copy a character-string.

SYNOPSIS

```
#include <string.h>
char *strcpy(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1	The address of the target buffer
s2	The address of the character-string to copy

Returns

<s1>

DESCRIPTION

This function copies the character-string referenced by <s2> into the buffer whose address is <s1>. It returns the address of the target buffer as its result.

NOTES

The result of this function is always a null-terminated character-string. The standard C library does not define the behavior of overlapping data movement, so using overlapping data movement may result in differing behavior on different systems. The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strncpy()

strcspn

Determine the unlike character-count.

SYNOPSIS

```
#include <string.h>
int strcspn(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1	The address of the character-string to examine
s2	The address of the character-string containing the characters to search for

Returns

The length of the initial segment of <s1> containing none of the characters found in <s2>

DESCRIPTION

This function examines the character-string whose address is <s1> and determines the length of the initial character segment containing none of the characters found in the character-string whose address is <s2>. It returns this count as its result.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strspn()

strlen

Determine the length of a character-string.

SYNOPSIS

```
#include <string.h>
int strlen(s)
char     *s;
```

Arguments

s¦ The address of a character-string

Returns

The number of characters in the character-string

DESCRIPTION

This function determines the length of character-string referenced by <s> and returns that value as its result.

It determines the length of the character-string by counting the number of characters which are not null-characters beginning at the address <s>, continuing until a null-character is found.

NOTES

A null-string ("") has a length of zero. The include-file "<string.h>" defines the string-handling functions in the C library.

strncat

Concatenate one character-string onto another.

SYNOPSIS

```
#include <string.h>
char *strncat(s1, s2, n)
char     *s1;
char     *s2;
int       n;
```

Arguments

s1	The address of the target character-string
s2	The address of the character-string to concatenate onto <s1>
n	The maximum number of characters to concatenate

Returns

<s1>

DESCRIPTION

This function appends at most <n> characters from the character-string referenced by <s2> onto the character-string referenced by <s1>. It re- turns as its result <s1>.

NOTES

The function always appends a null-character onto the characters appended onto <s1> from <s2>. The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strcat()

strncmp

Compare two character-strings.

SYNOPSIS

```
#include <string.h>
int strncmp(s1, s2, n)
char     *s1;
char     *s2;
int       n;
```

Arguments

s1	The address of the first string to compare
s2	The address of the second string to compare
n	The maximum number of characters to compare

Returns

A value less than, equal to, or greater than zero, if the first <n> characters in the character-string referenced by <s1> is lexicographically less than, equal to, or greater than the first <n> characters in the character-string referenced by <s2>

DESCRIPTION

This function lexicographically compares a maximum of <n> characters from the character-string referenced by <s1> with a maximum of <n> characters of the character-string referenced by <s2> and returns as its result a value which indicates the result of that comparison. That value is less than, equal to, or greater than zero, indicating that the first <n> characters of the character-string referenced by <s1> is lexicographically less than, equal to, or greater than the fist <n> characters of the character-string referenced by <s2>.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strcmp()

strncpy

Copy a character-string.

SYNOPSIS

```
#include <string.h>
char *strncpy(s1, s2, n)
char     *s1;
char     *s2;
int       n;
```

Arguments

s1	The address of the target buffer
s2	The address of the character-string to copy
n	The maximum number of characters to copy

Returns

<s1>

DESCRIPTION

This function copies characters from the character-string referenced by <s2> into the buffer whose address is <s1> until a null-character is copied or <n> characters have been copied, which ever comes first. It returns the address of the target buffer as its result.

NOTES

The function does not append a null-character to the copied characters. The standard C library does not define the behavior of overlapping data movement, so using overlapping data movement may result in differing behavior on different systems. The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strcpy()

strpbrk

Find the first occurrence of any of a list of characters in a character-string.

SYNOPSIS

```
#include <string.h>
char *strpbrk(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1¦ The address of the character-string to search
s2¦ The address of the character-string containing the list of characters to search for

Returns

The address of the first occurrence of any of the characters in <s2> found in <s1>, or (char *) NULL if none of the characters in <s2> was found in <s2>

DESCRIPTION

This function searches the character-string whose address is <s1> for the first occurrence of any character in the character-string whose address is <s2> and returns as its result the address of that character in <s1>. If the function fails to find any of the characters in <s2> in the character-string <s1> it returns (char *) NULL as its result.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strchr()

strrchr

Find the last occurrence of a character in a character-string.

SYNOPSIS

```
#include <string.h>
char *strrchr(s, c)
char     *s;
char      c;
```

Arguments

s¦ The address of the character-string to search
c¦ The character to search for

Returns

The address of the last occurrence of the character in the string, or (char *) NULL if the string does not contain the character

DESCRIPTION

This function searches the character-string whose address is <s> for the last occurrence of the character <c>. If the string contains the char- acter, the function returns as its result the address of the last oc- currence of the character. Otherwise, it returns (char *) NULL.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: rindex(), strchr()

strspn

Determine the like character-count.

SYNOPSIS

```
#include <string.h>
int strspn(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1¦ The address of the character-string to examine
s2¦ The address of the character-string containing the characters to search for

Returns

The length of the initial segment of <s1> containing only characters found in <s2>

DESCRIPTION

This function examines the character-string whose address is <s1> and determines the length of the initial character segment containing only characters found in the character-string whose address is <s2>. It re- turns this count as its result.

NOTES

The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strcspn()

_strtoi

Convert the digits in a character-string to an int.

SYNOPSIS

```
int _strtoi(str, ptr, base)
char     *str;
char    **ptr;
int       base;
```

Arguments

str| The address of the character-string to convert to an integer

ptr| The address of the char * to contain the address of the character which terminates the conversion, or (char **) NULL if none

base| The base of the digits

Returns

The value generated from the character-string.

DESCRIPTION

This function converts the character-string referenced by <str> to an int. It considers the digits to be in the base specified by <base> and assigns the address of the character ending the conversion to the char * referenced by <ptr>. The character that ends the conversion is either the null-character terminating the string or the first character that was inconsistent with the base. If <ptr> is (char **) NULL, the function does not make this assignment.

If the argument <base> is greater than 0 and less than or equal to 36, that value the base of the digits in the character-string. (For bases between 11 and 36, the alphabetic characters 'A' through 'Z' inclusive, in lexicographic order, are the digits of the base. The function considers lower-case characters to be the same as uppercase characters.)

If the base is 0, the function examines the character-string to determine the base. If following the optional white-space and sign is "0x" or "0X", the base is assumed to be 16. Otherwise, if a '0' follows the optional white-space and sign, the base is assumed to be 8. Otherwise, the base is assumed to be 10. If the base is less than 0 or greater than 36, the base is assumed to be 10.

NOTES

The function ignores overflow conditions.

SEE ALSO

C Library: _atoh(), atoi(), atol(), _atoo(), _atos(), strtol()

strtok

Extract the next token from a character-string.

SYNOPSIS

```
#include <string.h>
char *strtok(s1, s2)
char     *s1;
char     *s2;
```

Arguments

s1| The address of the character-string to search, or (char *) NULL

s2| The address of the character-string containing the token separators

Returns

The address of the first character of the next token, or (char *) NULL if none

DESCRIPTION

If the argument <s1> is not (char *) NULL, this function begins scanning with the first character in the character-string whose address is <s1> for the first character which is not in the token-separator character-string whose address is <s2>. Otherwise, the function begins scanning at the continuation-address set by a previous call, if any.

If the function finds no characters which are not token-separators, the function sets the continuation-address to (char *) NULL and returns (char *) NULL as its result. Otherwise, it remembers the address of that charac- ter, as the value the function will return as its result and continues scanning, looking for the next character that is a token-separator.

If it finds a token-separator, it changes that character to a null-character (' 0'), sets the continuation-address to that of the character following that token-separator. Otherwise, it sets the continuation-address to (char *) NULL. The function then returns the remembered address, the address of the token, as its result.

NOTES

The function always returns (char *) NULL if it is called with the first argument (char *) NULL and there is no continuation-address. There is no continuation address if the function has not been called with the first argument something other than (char *) NULL or the function returned (char *) NULL the last time it was called. The separator string referenced by <s2> need not be the same string from one call to this function to another.

If the function returns as its result something other than (char *) NULL, that result always references a character-string (a null-terminated string of characters). The include-file "<string.h>" defines the string-handling functions in the C library.

SEE ALSO

C Library: strchr(), strpbrk(), strrchr()

strtol

Convert the digits in a character-string to a long.

SYNOPSIS

```
long strtol(str, ptr, base)
char     *str;
char    **ptr;
int       base;
```

Arguments

str¦ The address of the character-string to convert to an integer

ptr¦ The address of the char * to contain the address of the character which terminates the conversion, or(char **) NULL if none.

base¦ The base of the digits

Returns

The value generated from the character-string.

DESCRIPTION

This function converts the character-string referenced by <str> to a long. It considers the digits to be in the base specified by <base> and assigns the address of the character ending the conversion to the char * referenced by <ptr>. The character that ends the conversion is either the null-character terminating the string or the first character that was inconsistent with the base. If <ptr> is (char **) NULL, the function does not make this assignment.

If the argument <base> is greater than 0 and less than or equal to 36, that value the base of the digits in the character-string. (For bases between 11 and 36, the alphabetic characters 'A' through 'Z' inclusive, in lexicographic order, are the digits of the base. The function considers lowercase characters to be the same as uppercase characters.)

If the base is 0, the function examines the character-string to determine the base. If following the optional white-space and sign is "0x" or "0X", the base is assumed to be 16. Otherwise, if a '0' follows the optional white-space and sign, the base is assumed to be 8. Otherwise, the base is assumed to be 10. If the base is less than 0 or greater than 36, the base is assumed to be 10.

NOTES

The function ignores overflow conditions.

SEE ALSO

C Library: _atoh(), atoi(), atol(), _atoo(), _atos(), _strtoi()

timezone

Current time zone value.

SYNOPSIS

```
#include <time.h>
extern long timezone;
```

DESCRIPTION

This variable contains the current time zone value which is the number of seconds the zone is west of (behind) Greenwich Mean Time (Universial Coordinated Time).

This variable is initialized automatically by "localtime()" and "ctime()" and may be initialized explicitly by "tzset()". Before initialization, the value is zero.

NOTES

The include-file "<time.h>" defines this external variables along with other external variables and functions.

SEE ALSO

C Library: ctime(), daylight, localtime(), tzname, tzset()

toascii

Generate a value that is within the range of valid ASCII characters.

SYNOPSIS

```
#include <ctype.h>
int toascii(c)
int  c;
```

Arguments

c¦ Value to be examined

Returns

<c> & 0x7F

DESCRIPTION

This function generates a value that is within the range of ASCII characters from the value <c> and returns that value as its result. It does this by anding the value <c> with the bit-string 0x7F (127). The result is a value between 0x00 and 0x7F inclusive, which is the range of ASCII characters.

NOTES

The argument <c> will be cast into an int if it is not already of that type. The include-file "<ctype.h>" defines this function and other functions which test and manipulate characters. It must be included in the C source before the first reference to this function.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), _tolower(), toupper(), _toupper()

_tolower

Convert an upper-case character to a lower-case character.

SYNOPSIS

```
#include <ctype.h>
int _tolower(c)
int  c;
```

Arguments

c¦ Value to convert

Returns

The converted value

DESCRIPTION

This function converts an uppercase alphabetic ASCII character to its equivalent lowercase alphabetic character and returns that value as its result.

NOTES

This function is implemented as a macro. It will have no side-effects but the result of the function is defined only for values of <c> which are upper-case alphabetic ASCII characters. The argument <c> will be cast into an int if it is not already of that type. The include-file "<ctype.h>" defines this function and other functions which test and manipulate characters. It must be included in the C source program before the first reference to this function.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), toupper(), _toupper()

_toupper

Convert a lower-case character to an upper-case character.

SYNOPSIS

```
#include <ctype.h>
int _toupper(c)
int  c;
```

Arguments

c¦ Value to convert

Returns

The converted value

DESCRIPTION

This function converts a lowercase alphabetic ASCII character to its equivalent uppercase alphabetic character and returns that value as its result.

NOTES

This function is implemented as a macro. It has no side-effects but its result is only defined for values of <c> which are lowercase alphabetic ASCII characters. The argument <c> will be cast into an int if it is not already of that type. The include-file "<ctype.h>" defines this function and other functions which test and manipulate characters. It must be included in the C source before the first reference to this function.

SEE ALSO

C Library: isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), toascii(), tolower(), _tolower(), toupper()

ttyname

Generate the pathname for a terminal.

SYNOPSIS

```
char *ttyname(fildes)
int       fildes;
```

Arguments

fildes¦ A file descriptor for the terminal

Returns

The address of a character-string containing a pathname for the terminal or (char *) NULL if <fildes> is not a file descriptor for a terminal residing in the directory "/dev"

DESCRIPTION

This function determines if the file referenced by the file descriptor <fildes> is a character-special file (a terminal), and is reached by a pathname whose path is the directory "/dev". If the file satisfies these conditions, the function generates a complete pathname for the file and returns as its result the address of a character-string containing that complete pathname. Otherwise, it returns as its result (char *) NULL.

NOTES

The character-string addressed by the result of this function is in static memory and is overwritten by subsequent calls to this function. A file descriptor is an index into the operating system's open file table. The system functions "creat()", "dup()", "dup2()", "open()", and "pipe()" return a file descriptor as their result. The function "fileno()" determines the file descriptor of a stream.

SEE ALSO

C Library: fileno(), isatty()

System Call: creat(), dup(), dup2(), open(), pipe(), ttyslot()

tzname

Time-zone name abbreviations.

SYNOPSIS

```
#include <time.h>
extern char *tzname[2];
```

DESCRIPTION

This external variable is two-element array of references to character-strings. The first element references the three-character abbreviated name of the standard-time time zone, contained in a character-string. The second references the three-character abbreviated name of the daylight-time time zone, contained in a character-string. A (char *) NULL value indicates that "tzname" has not been initialized. A null-string indicates that the abbreviated name is not known.

The list is initialized automatically by "localtime()" and "ctime()" and may be initialized explicitly by "tzset()". The values in the list are (char *) NULL before initialization.

SEE ALSO

C Library: ctime(), daylight, localtime(), timezone, tzset()

tzset

Initialize external variables containing time parameters.

SYNOPSIS

void tzset()

Arguments

None

Returns

Void

DESCRIPTION

This function initializes the global variables "daylight", "timezone", and "tzname" according to the current system configuration.

The variable "daylight" is non-zero if the standard U.S.A. daylight-savings time conversion is to be applied to all time conversions from system time or Greenwich Mean Time (GMT) to the time in the local time zone, otherwise it is zero. The variable "timezone" contains the number of seconds the current time zone is west of GMT. The array "tzname" contains two elements, the first is the address of a character-string containing the abbreviation for the current standard time-zone name, the second is the address of a character-string containing the abbreviation for the current daylight time-zone name. If one or the other is not known, the strings will be null-strings.

NOTES

This function is called automatically by "localtime()" and "ctime()".

SEE ALSO

C Library: ctime(), daylight, localtime(), timezone, tzname

ungetc

Push a character onto an input stream.

SYNOPSIS

```
int ungetc(c, stream)
int       c;
FILE     *stream;
```

Arguments

c¦	The character to push onto the stream
stream¦	The stream to get the character

Returns

Its argument <c> or EOF

DESCRIPTION

If <c> does not equal EOF, this function pushes (char) <c> onto the standard I/O input stream <stream>. If the function succeeds, it returns its argument <c>, otherwise it returns EOF.

NOTES

A stream may only have one character pushed onto it at a time. Attempting an "ungetc()" on a stream that already has a character pushed onto it results in losing the previously pushed character. The function returns EOF of the stream referenced by <stream> is not an input stream. The "fseek()" and "rewind()" functions undo the effects of this function. The result of "ftell()" does not reflect any character pushed onto the stream.

SEE ALSO

C Library: fdopen(), fgetc(), fopen(), fseek(), getc(), rewind()

THE GRAPHICS LIBRARY

The graphics library provides access to the bit-mapped display and to the event manager. It uses the mechanisms that support Smalltalk's use of the bit-mapped display, the keyboard, and the mouse. The graphics library allows applications to use the "bitblt" graphics primitive, change the cursor, detect button presses, and perform simple graphics operations such as draw lines and boxes, as well as other related abilities.

The library itself is in the file named "/lib/graphics", with a C header file which defines the various structures in "/lib/include/graphics.h".

The following conventions apply to the graphics library:

- All arguments are of the type int or pointer to int, unless otherwise specified.
- In all the following descriptions, the C language definitions of true and false are valid.
- For true/false arguments are interpreted by the functions as true <> 0 and false == 0.
- The values returned from the library functions should be interpreted as true > 0, false == 0, and error condition < 0.
- All functions without explicit return values will return success/failure indications as success == 0, failure (error condition) < 0.
- Any function which returns an error condition will also set the global variable "errno" to an appropriate error code.

The following routines are included in the graphics library:

```
struct FORM *InitGraphics(arg)
```

Map the bit-mapped display into the address space of the calling process and put the display in graphics mode. If the argument is false, all other modes are unchanged. If the argument is true, then in addition, the display is cleared, made visible, and set to normal video (black on white) with both mouse and joydisk panning enabled. This routine returns a pointer to a form which defines the screen bit-map.

```
ExitGraphics()      Terminate use of graphics mode.
```

The display may be re-initialized by the terminal emulator.

```
ClearScreen()       Set the full screen bit-map to zeros.
```

bool

If the screen is set to normal video, this will result in a white screen. The terminal emulator is not affected by this call, i.e., the terminal emulator's idea of where to place its next character is unchanged.

```
struct FORM *FormCreate(w, h)
```

Allocate memory for a bit-map of the specified size and return a pointer to a form which defines it.

```
FormDestroy(struct FORM *form)
```

Free a form and the bit-map associated with it. The screen form, returned by InitGraphics, cannot be destroyed.

```
DisplayVisible(arg)
```

Make the display visible or blanked. The display is made visible if the argument is true, otherwise it is blanked. The previous mode is returned (true for visible, false for blanked).

```
CursorVisible(arg)
```

Make the cursor visible or invisible. The cursor is made visible if the argument is true, otherwise it is blanked. The previous mode is returned (true for visible, false for invisible).

```
VideoNormal(arg)
```

Set the video mode of the display. The mode is set to normal video (black on white) if the argument is true, and to inverse video (white on black) otherwise. The previous mode is returned (true for normal, false for inverse).

PanCursorEnable(arg)

Enable screen panning using the cursor. If the argument is true then auto-panning with the cursor is enabled, otherwise it is disabled. The previous mode is returned (true for cursor auto-panning enabled, false for cursor auto-panning disabled).

PanDiskEnable(arg)

Enable screen panning using the joydisk. If the argument is true then auto-panning with the joydisk is enabled, and otherwise it is disabled. The previous mode is returned (true for joydisk auto-panning enabled, false for joydisk auto-panning disabled).

ScreenSaverEnable(arg)

Enable the screen saver timeout, which causes the screen to be blanked after 10 minutes of keyboard or mouse inactivity. If the argument is true then the timeout is enabled, otherwise it is disabled. The previous mode is returned (true for screen saver enabled, false for screen saver disabled).

TerminalEnable(arg)

Enable the terminal emulator. If the terminal emulator is disabled, no process can print characters on the screen unless it calls PaintString. If the argument is true then the terminal emulator is enabled, otherwise it is disabled. The previous mode is returned (true for terminal emulator enabled, false for terminal emulator disabled).

CursorTrack(arg)

Force the cursor to track the mouse. If the argument is true, then moving the mouse will cause the cursor to track the mouse position, otherwise the mouse will have no affect on the cursor. The previous mode is returned (true for track, false for non-tracking).

SetViewport(struct POINT *point)

Set the panning hardware to display the upper left-hand corner of the 640x480 display at the specified position. The x and y values defined in the point must be in the physical range of the screen.

GetViewport(struct POINT *point)

Get the position which the panning hardware is displaying as the upper left-hand corner of the 640x480 display. The x and y values returned in the point will be in the range 0 to 1023, inclusive.

SetCursor(struct FORM *cursor)

Install a new cursor. The cursor parameter points to a form containing the 16x16 bit representation for the new cursor.

GetCursor(struct FORM *cursor)

Return the current cursor image. The cursor parameter must point to a 16x16 bit form. That form will have the 16x16 bit representation of the current cursor placed in it.

SetCPosition(struct POINT *point)

Display the cursor at the specified position. If cursor/mouse tracking is enabled, i.e., CursorTrack(TRUE), this is the same as SetMPosition. The x and y values defined in the point must be in the range 0 to 1023, inclusive.

GetCPosition(struct POINT *point)

Get the position where the cursor is currently displayed. If cursor/mouse tracking is enabled, i.e., CursorTrack(TRUE), this is the same as GetMPosition. The x and y values returned in the point will be in the range 0 to 1023, inclusive.

SetMPosition(struct POINT *point)

Position the mouse at the specified position. If cursor/mouse tracking is enabled, i.e., CursorTrack(TRUE), this is the same as SetCPosition. The x and y values defined in the point must be in the range 0 to 1023, inclusive.

GetMPosition(struct POINT *point)

Get the position where the mouse is currently pointing. If cursor/mouse tracking is enabled, i.e., CursorTrack(TRUE), this is the same as DGetCPosition. The x and y values returned in the point will be in the range 0 to 1023, inclusive.

SetMBounds(struct POINT *p1, struct POINT *p2)

Set the limits on mouse motion to be the rectangle defined by the upper left point p1 and the lower right point p2. The x and y values defined in the point may be in the range -32768 to 32767, inclusive.

GetMBounds(struct POINT *p1, struct POINT *p2)

Get the limits on mouse motion. The x and y values returned in the point will be in the range -32768 to 32767, inclusive.

SaveDisplayState(struct DISPSTATE *dp)

Copy significant parameters of the current display state into the structure designated. These parameters will include at least the coordinates of the viewport, the mouse bounds, the current cursor, the keyboard code, and the display, terminal emulator, cursor, panning, tracking, screen saver, and video modes.

RestoreDisplayState(struct DISPSTATE *dp)

The state defined by the argument structure is re-established.

ProtectCursor(struct RECT *r1, struct RECT *r2)

Tell the operating system that graphics operations will be occurring in one or both of the screen areas defined by the two rectangles (either rectangle pointer may be null). The operating system will respond by removing the cursor from the screen if it is in either of the two areas. This instruction and its release (ReleaseCursor) should be used if the user is writing or reading directly from the screen. This cursor protection is already included in the routines of this library which draw on the screen.

ReleaseCursor()

Tell the operating system to restore the cursor if it was removed due to a ProtectCursor call. This call should be used to match every ProtectCursor call.

GetButtons()

Return an int which indicates the state of the mouse buttons. The buttons states are reported in the low three bits of the int, where bit 0 is the right button, bit 1 is the middle buttons, and bit 2 is the left button. If the bit is a 0 then the button is up, if the bit is a 1 then the button is down (depressed). The button bits are defined constants in the graphics header file.

BitBlt(struct BBCOM *bitbltComPtr)

Perform the bitblt command described in the record pointed to by the parameter. The record contains the source and destination rectangles, clipping regions, halftone mask, and combination rule.

PaintLine(struct BBCOM *bitbltComPtr, struct POINT *p)

Paint a line on the display. A sequence of bitblt operations is performed while stepping a pixel at a time toward the specified position. If the one of the exclusive OR rules is specified, and the source is null (ones), the response will instead be as if the line was drawn by the above stepping method to a hidden bit-map and then that hidden bit-map was combined with the destination bit-map according to the specified rule.

PointToRC(int *row, int *col, struct POINT *p)

Convert a screen coordinate to the row, column indices which define the terminal emulator character cell which contains that point.

RCToRect(struct RECT *rectp, int row, int col)

Given row and column indices which define a terminal emulator character cell, returns the rectangle which describes that cell.

EventEnable()

Enables event processing, i.e., turns on the event manager. Any subsequent user input action will cause event values to be created. Keyboard input through the "console" device and terminal emulator is disabled.

EventDisable()

Disables event processing, i.e., turns off the event manager. Keyboard input through the "console" device and terminal emulator is re-enabled.

ESetSignal()

Request the event manager to signal the current process when events occur. The event signal is disabled after being issued.

ESetMInterval(freq)

Specifies how frequently mouse motion events are to be created if the mouse is continuously moving.

EGetCount()

Returns the number of event values in the event buffer waiting to be processed. Return of a negative number indicates an error.

EGetNewCount()

Returns the number of event values in the event buffer which have occurred since the previous call to this function.

union EVENTUNION EGetNext()

Returns the next event value in the event buffer. Since some event values are self-contained and some are headers to the following values, this function returns either a structure or an unsigned short.

unsigned long EGetTime()

Returns the time, in milliseconds, since the system was powered up. A return time of 0 indicates an error.

ESetAlarm(unsigned long time)

Requests a signal when the specified time, in milliseconds, is reached.

EClearAlarm()

Clears any pending alarms that the process has requested.

SetKBCode(arg)

This tells the keyboard to output either event codes, if arg is 0, or ANSI character strings, if arg is 1. Other arguments are not recognized at this time. Event processing must be enabled before asking for event codes, and enabling events automatically forces the keyboard into event mode. This would normally be used after enabling the event mechanism, to force the keyboard back to ANSI mode, while leaving the mouse generating events.

#INCLUDE FILES

The directory "/lib" contains a subdirectory "/lib/include/", which contains the subdirectory "/lib/include/sys". In these directories, you will find the files that are available to the "#include" preprocessor command.

Section 8

4404 HARDWARE SUPPORT

INTRODUCTION

You can access the 4404 hardware directly, but in general, this tends to be cumbersome and error-prone. The operating system has embedded within it device drivers, or software routines that offer a uniform interface with the operating system. Most programs should interface with the 4404 through these device drivers.

This section discusses the device drivers and system calls to the 4404 hardware.

DEVICE DRIVERS

Device drivers are divided into two types: block-oriented and character-oriented. Block-oriented devices include the disks and other peripherals on the SCSI bus. Character-oriented devices include the console, the communications port, the sound generator, the printer port, the optional network interface, and special devices for "raw" access to the block-oriented devices. Each of these devices is identified by a file in the "/dev" directory.

The system calls "ttyget" and "ttyset" can be used with the console, communications port, sound, and printer devices. Descriptions of the parameters to these calls are found in Section 6, System Calls.

SCSI PERIPHERALS

A SCSI bus gives access to the block-oriented devices. These devices include such things as winchester disks, floppy disks, and (optional) streaming tape drives.

The /dev SCSI peripheral devices are:

- disk -- The winchester disk with the system files.
- disk1..diskn -- Optional winchester disks.
- floppy -- Floppy disk drive.

The standard 4404 contains a single floppy disk (/dev/floppy) and a 40 Megabyte winchester (/dev/disk). Option 20 contains an additional 40 Megabyte winchester disk and a streaming tape drive.

Device "/dev/disk" is the standard system device and is the default device from which to boot the system. You must use the interactive boot procedure to boot from another device.

CONSOLE DEVICE

The device "/dev/console" supports the 4404 display and keyboard. It is connected to a terminal emulator which acts like an industry-standard terminal (described in Section 10 of this manual).

To read and write terminal settings and other parameters of this device, use the "ttyget" and "ttyset" system calls, or the "conset" utility.

COMMUNICATIONS PORT

The device "/dev/comm" supports the RS-232C host communications port. You can control the baud rate, number of stop bits, parity, and XON/XOFF or DTR flow control. You can also cause new input or completion of output to generate a signal, as well as read the number of characters pending in the input and output queues.

The "ttyget" and "ttyset" system calls, and the "commset" utility allow you to read and write the communications port parameters.

SOUND GENERATOR

The device "/dev/sound" is the 4404 sound generator. By sending a formatted byte stream to this device (a TI 76496 sound generator chip), you can cause the 4404 to produce sounds.

This device is a write-only device. An attempt to read it will return an error. It is also an exclusive-use device and may be opened by only one task at a time.

The "ttyset" and "ttyget" system calls can change operation settings and examine device status.

Controlling the Sound Device

"/dev/sound" expects a stream of bytes in the following form:

```
n,c,c,c...c,t
or
0,tempo
```

with the following values:

n -- A single byte specifying the number of commands to follow.

c -- A single byte binary command to the sound chip. (See the following discussion on sound chip operation and commands.)

t -- A byte value specifying the length of time to hold this set of commands. T is in units of tempo set by the second format.

tempo -- A 16-bit (word) value of time with a unit value of 16.667 ms.

"/dev/sound" Operation and Commands

The sound chip contains three frequency generators, each coupled to a programmable attenuator. It also contains a white-noise generator (a shift register with an exclusive-OR feedback network) that contains a frequency control and programmable attenuator.

Frequency Control. Changing the value in a frequency generator requires two command bytes. Byte 1 contains the address information (which frequency generator to alter) and the low order 4 bits of the value to store. Byte 2 contains the high order 6 bits to set the frequency. Thus, the two bytes contain a 3-bit address and a 10-bit binary number to set the frequency to be generated.

The frequency is equal to the clocking rate of the chip (which is 3.15 MHz) divided by thirty-two times the binary number that is stored in the frequency generator.

Table 8-1 shows the bit assignments in Byte 1 and Table 8-2 shows the bit assignments in byte 2.

Table 8-1

FREQUENCY SELECTION (BYTE 1)

Bit	Type	Description
0	1	This bit is always 1
1	R2	Register address bit 2
2	R1	Register address bit 1
3	R0	Register address bit 0
4	F3	Frequency data bit 3
5	F2	Frequency data bit 2
6	F1	Frequency data bit 1
7	F0	Frequency data bit 0

Table 8-2

FREQUENCY SELECTION (BYTE 2)

Bit	Type	Description
0	0	This bit is always 0
1	x	Unused
2	F9	Frequency data bit 9
3	F8	Frequency data bit 8
4	F7	Frequency data bit 7
5	F6	Frequency data bit 6
6	F5	Frequency data bit 5
7	F4	Frequency data bit 4

Controlling Attenuation. You can control the attenuation on any frequency generator with a single command byte. This byte contains a 3-bit field to select the attenuator and a 4-bit field to specify the amount of attenuation. Four bits give you 16 possible attenuations. Table 8-3 shows the attenuation settings and table 8-4 shows the bit assignments for the attenuation control byte.

Table 8-3

ATTENUATION CONTROL

A3	A2	A1	A0	Attenuation Weight (dB)
0	0	0	0	ON (Full Volume)
0	0	0	1	2
0	0	1	0	4
0	1	0	0	8
1	0	0	0	16
1	1	1	1	Off

Table 8-4

ATTENUATION BYTE BIT ASSIGNMENTS

Bit	Type	Description
0	1	Always 1
1	R2	Register address bit 0
2	R1	Register address bit 1
3	R0	Register address bit 2
4	A3	Attenuation control bit 3
5	A2	Attenuation control bit 2
6	A1	Attenuation control bit 1
7	A0	Attenuation control bit 0

Controlling the Noise Generator. The noise generator consists of a noise source and an attenuator. You can control the type of feedback in the exclusive-OR network, the shift rate, and the attenuator itself.

Table 8-5 shows how you control the feedback, table 8-6 shows the shift-rate control, and table 8-7 shows the bit assignments for the noise generator command byte.

Table 8-5

NOISE FEEDBACK CONTROL

FB	Configuration
0	Periodic noise
1	White noise

Table 8-6

NOISE FREQUENCY CONTROL

NF1	NF0	Shift Rate
0	0	49218
0	1	24609
1	0	12304
1	1	Tone generator #3 output

Table 8-7

NOISE-CONTROL-BYTE BIT ASSIGNMENTS

Bit	Type	Description
0	1	Always 1
1	R2	Register address bit 2
2	R1	Register address bit 1
3	R0	Register address bit 0
4	x	Unused
5	FB	Feedback control bit
6	NF1	Shift rate control bit 1
7	NF2	Shift rate control bit 0

Control Registers. The sound chip has eight internal registers that determine whether the byte(s) sent control the frequency or attenuation of the three tone generators or the control or attenuation of the noise generator. The destinations for all addressed bytes are given in Table 8-8.

Table 8-8

CONTROL REGISTER ADDRESSES

R2	R1	R0	Address register
0	0	0	Tone 1 frequency
0	0	1	Tone 1 attenuation
0	1	0	Tone 2 frequency
0	1	1	Tone 2 attenuation
1	0	0	Tone 3 frequency
1	0	1	Tone 3 attenuation
1	1	0	Noise control
1	1	1	Noise attenuation

SOUND EXAMPLES

The following examples show how you can control the sound device by sending bytes to it. You can have a program send the bytes directly to the sound device, "/dev/sound", or you can have your program store these bytes into a file, "noiseFile", then issue the command:

```
list noiseFile >/dev/sound
```

Set the tempo to be 1 beat per second (1000 millisec/beat)

Byte 1 = 0 "Tempo word follows"

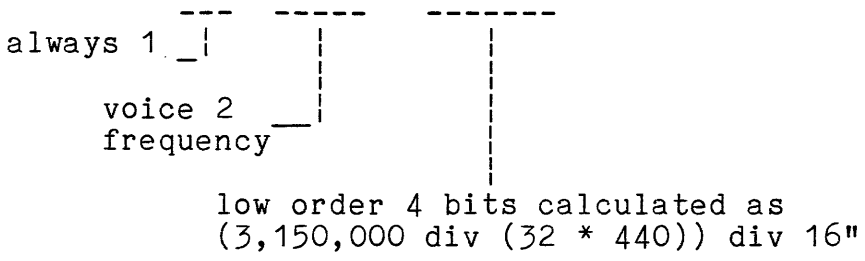
Byte 2 = 0 "high order byte = (1000 div 16.667) div 256"

Byte 3 = 59 "low order byte = (1000 div 16.667) mod 256"

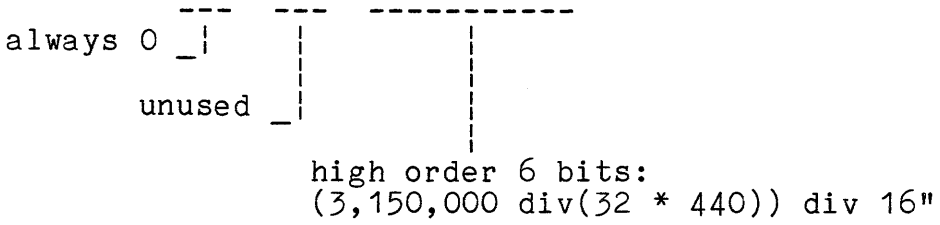
Set the frequency for voice 2 to be 440 Hz

Byte 4 = 2 "2 command bytes follow"

Byte 5 =175 "1 0 1 0 1 1 1 1"



Byte 6 = 13 "0 0 0 0 1 1 0 1

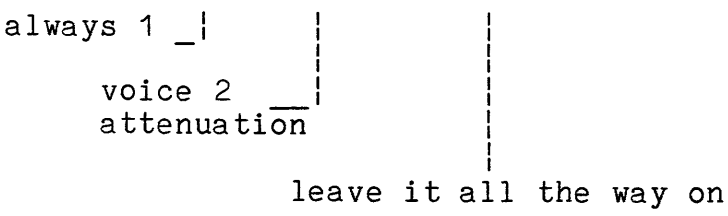


Byte 7 = 0 "hold this set of commands 0 beats"

Play voice 2 at full volume for 1 beat

Byte 8 = 1 "1 command follows"

Byte 9 =176 "1 0 1 1 0 0 0 0

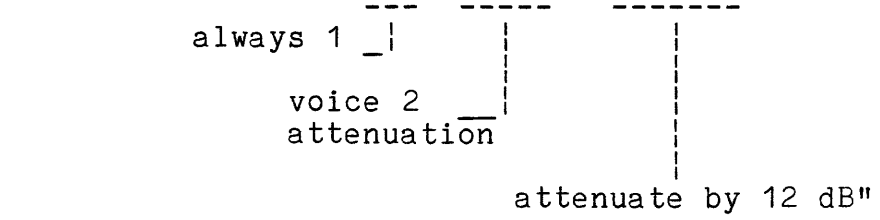


Byte 10 = 1 "play for 1 beat"

Turn the volume of voice 2 down by 12 dB and play for 2 beats

Byte 11 = 1 "1 command byte follows"

Byte 12 =188 "1 0 1 1 1 1 0 0"

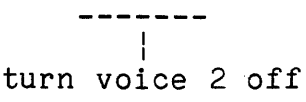


Byte 13 = 2 "hold for 2 beats"

Turn voice 2 off so it won't play forever"

Byte 14 = 1 "1 command byte follows"

Byte 15 =191 "1 0 1 1 1 1 1 1"

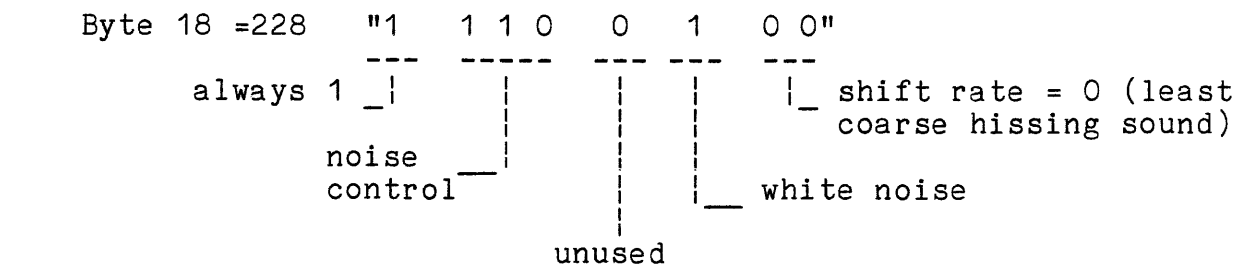


Byte 16 = 1 "hold for 1 beat"

Play white noise (hissing sound)

Byte 17 = 1 "1 command byte follows"

Byte 18 =228 "1 1 1 0 0 1 0 0"

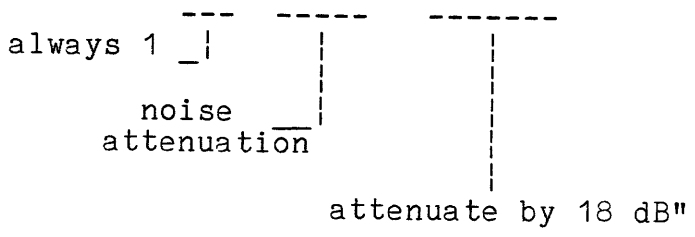


Byte 19 = 0 "hold 0 beats"

Turn down the volume 18 dB and hold for 2 beats

Byte 20 = 1 "1 command follows"

Byte 21 =249 "1 1 1 1 1 0 0 1"

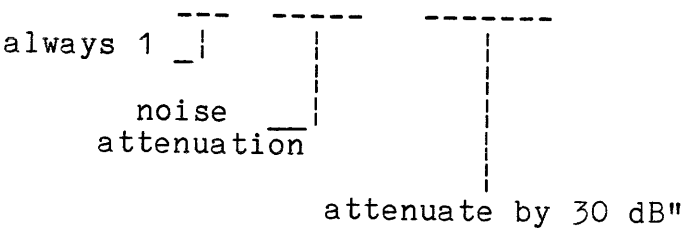


Byte 22 = 2 "hold 2 beats"

Turn noise off

Byte 23 = 1 "1 command follows"

Byte 21 =255 "1 1 1 1 1 1 1 1



Byte 25 = 1 "hold 1 beat"

PRINTER PORT

The device "/dev/printer" provides interface to the 4404 parallel interface printer port. This port provides a Centronics-compatible parallel port that can drive most inexpensive dot-matrix (and some letter quality) printers.

"/dev/printer" accepts character streams and recognizes the ANSI X3.64 "Select Graphic Rendition" escape sequences for bold or italic characters.

These devices are write-only; any attempt to read them will return an error. They are exclusive-use devices and may be opened by only one task at a time.

The system calls "ttyget" and "ttyset" can be used to examine device status and change operation settings.

OTHER DEVICES

The /dev directory contains other entries for devices supported by the operating system:

disk_c	Raw system disk
disk1_c..diskn_c	Raw optional winchester disk
floppy_c	Raw floppy disk drive
tape_c	Raw streaming tape drive
null	Null device
pmem	Physical memory
smem	System memory
swap	Swap space on winchester disk

These devices are all character-oriented. The raw versions of the peripheral devices provide access to them as simple character streams without file systems. The null device may be used as a data sink. The memory devices can be used to inspect and modify the system's memory.

These devices, with the exception of /dev/null, are reserved for use by system programs.

CAUTION

> Use of these devices is an excellent way to crash the operating system and destroy the disk file structure.

DISPLAY, MOUSE, AND KEYBOARD SUPPORT

The 4404 display is a 1024 X 1024 virtual display viewed through a 640 X 480 physical display viewport. The operating system includes support that allows positioning and smooth panning of the viewport over the virtual display. The operating system also supports the creation and movement of a display cursor.

The 4404 display uses Smalltalk-80 conventions. The upper-left corner of the display has coordinates (0,0), while the lower-right corner has coordinates (1023,1023). X coordinates increase toward the bottom of the screen; Y coordinates increase to the right.

Full program access to interactive event processing is supported through system calls to an event manager. Events include mouse movements, and up/down transitions of the mouse, keyboard, and joydisk contacts. The design of the event mechanism is patterned closely after a similar mechanism described on pages 648-650 of the book Smalltalk-80: The Language and Its Implementation.

DISPLAY PANNING

The operating system allows the 640 X 480 hardware display viewport to be positioned anywhere on the virtual display as long as the upper left corner has an X-coordinate less than 383 and a Y-coordinate less than 539.

The operating system supports the panning of the viewport over the virtual display under control of the mouse and joydisk. When joydisk panning is enabled, pushing the top of the joydisk causes the Y-coordinate to decrease by 5 pixels during each vertical sync interrupt, while pushing the bottom causes it to increase a like amount. Pushing the left side of the joydisk causes the X-coordinate to decrease 5 pixels per interrupt; while pushing the right side of the joydisk causes it to increase. Joydisk panning ceases in a particular direction when the coordinate for that direction reaches zero or its maximum value.

The cursor remains at a fixed position on the virtual display while the viewport is panned by the joydisk. When cursor panning is enabled, moving the cursor will also cause the viewport to pan so that the cursor is always located within the physical viewport. This allows the mouse to pan the viewport position because the cursor position is usually linked to mouse movement.

CURSOR AND MOUSE TRACKING

The cursor is a 16 X 16 bit-map that is displayed by logically ORing it into the display bit-map. The contents of the area under the cursor are saved and they are restored when the cursor is moved. The operating system allows the cursor's position to track the motion of the mouse. When this feature is enabled, the operating system will automatically move the cursor whenever the mouse is moved.

The mouse position is not allowed to exceed certain bounds when the cursor is linked to the mouse. The default bounds are the virtual display coordinates. The user may change these bounds and allow the cursor to be moved off the virtual display.

DISPLAY ACCESS FUNCTIONS

The operating system provides access to display functions through the processor's trap instruction. To invoke these functions, load register D0 with the function code (or parameters for the function) and issue a trap #13 instruction.

On return, the carry bit will be clear if there were no errors. If an error occurs, the carry bit will be set and register D0 will contain an error code.

The display functions are:

Display Function 0: cursorOn

Displays the cursor. Returns 1 in D0 if the cursor was previously enabled, 0 if it was not.

Display Function 1: cursorOff

Suspends display of the cursor. Returns 1 in D0 of the cursor was previously enabled, 0 if it was disabled.

Display Function 2: cursorLink

Causes the cursor to track the mouse. Returns 1 in D0 if the cursor was previously linked, 0 if it was not.

Display Function 3: cursorUnlink

Breaks the links that caused the cursor to track the mouse. Returns 1 in D0 if the cursor and mouse were linked, 0 if not.

Display Function 4: cursorPanOn

Causes the viewport to pan when the cursor reaches an edge. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 5: cursorPanOff

Disables viewport panning via cursor movement. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 6: displayOn

Makes the display visible. Returns 1 in D0 if previously visible, 0 if blanked.

Display Function 7: displayOff

Blanks the display. Returns 1 in D0 if previously visible, 0 if blanked.

Display Function 8: joyPanOn

Turns on panning via joydisk. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 9: joyPanOff

Disconnects the joydisk from viewport panning. Returns 1 in D0 if panning were previously enabled, 0 if not.

Display Function 10: timeoutOn

Causes the screen to automatically blank if inactive for ten minutes. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 11: timeoutOff

Disables automatic blanking. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 12: blackOnWhite

Sets the display to Normal Video mode. Returns 1 in D0 if previously normal, 0 if inverted.

Display Function 13: whiteOnBlack

Sets the display to Inverse Video mode. Returns 1 in D0 if previously normal, 0 if inverted.

Display Function 14: terminalOn

Enables use of the terminal emulator with the display. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 15: terminalOff

Disables use of the terminal emulator with the display. Returns 1 in D0 if previously enabled, 0 if not.

Display Function 16: getMousePoint

The position of the mouse is returned as an (X,Y) pair in the high and low halves of register D0. If the cursor is linked to the mouse, this is the same as the mouse position.

Display Function 17: setMousePoint

The current mouse position is set to the position passed as an (X,Y) pair in the high and low halves of register D0. If the cursor is linked to the mouse, the cursor position is also set.

Display Function 18: getCursorPoint

The current cursor position is returned as an (X,Y) pair in the high and low halves of register D0. If the cursor is linked to the mouse, this is the same as the mouse position.

Display Function 19: setCursorPoint

The current cursor position is set to the position passed as an (X,Y) pair in register D0. If the mouse is linked to the cursor, the mouse position is also set.

Display Function 20: getButtons

The state of the mouse buttons is returned in register D0. Bit 0 corresponds to the left button, bit 1 to the middle button, and bit 2 to the right button. Zero in a bit indicates that the corresponding button is up, one indicates that it is pressed.

Display Function 21: setSource

The source rectangle for a bitBlt operation is passes as an argument in registers D1 and D2. It is encoded as:

upper-left-corner	(X0,Y0) in the high and low halves of register D1
lower-right-corner	(X1,Y1) in the high and low halves of register D2.

The operating system insures that the cursor is not displayed in this area.

Display Function 22: setDest

The destination rectangle for a bitBlt operation is passed as an argument in registers D1 and D2. It is encoded as:

upper-left-corner	(X0,Y0) in the high and low halves of register D1
lower-right-corner	(X1,Y1) in the high and low halves of register D2.

The operating system insures that the cursor is not displayed in this area.

Display Function 23: updateComplete

This function allows the cursor to be displayed in areas previously specified as source or destination rectangles.

Display Function 24: getCursorform

This function gets the cursor (a 16 X 16 pixel bit-map stored as sixteen consecutive words). You must pass a pointer to this bit-map in register A0.

Display Function 25: setCursorform

This stores the image of the cursor (as a 16 X 16 bit-map of sixteen consecutive words) beginning at the address passed as a pointer in register A0.

Display Function 26: getViewport

Returns the position of the upper left corner of the physical 640 X 480 physical viewport in the 1024 X 1024 virtual display. This position is returned as an (X,Y) pair in the high and low halves of register D0.

Display Function 27: setViewport

Sets the display hardware to start updating from a specific position within the display bit-map. The position is specified as an X,Y pair passed in the high and low halves of register D1. This is used to position the 640x480 viewport anywhere within the 1024x1024 virtual display.

Display Function 28: getDisplayState

The current state of the display is returned in a record pointed to by register A0. The display state area must be at least ?? bytes in length (at an even address) and will contains the following information: <<to be supplied>>

Display Function 29: setKeyboardCode

The form of output generated by the keyboard is set by the value passed in D1. Valid values are:

0 sets keyboard output to event codes

1 sets keyboard output to ANSI terminal code sequences.

This call is normally only used after a "Enable Event Processing" call which implicitly sets the keyboard code to 0 (event codes). The previous keyboard code is return in D0.

Display Function 30: getMouseBounds

Return the limits of the rectangle within which the mouse and cursor are constrained in D0 and D1. D0 contains the coordinates of the upper left corner of the rectangle. D1 contains the coordinates of the lower right corner.

Display Function 31: setMouseBounds

Set the limits of the rectangle within which the mouse and cursor are constrained. D1 contains the coordinates of the upper left corner of the rectangle. D2 contains the coordinates of the lower right corner.

Display Function 32: XYtoRC

Convert screen coordinates to terminal row and column. D1 contains the coordinates of a point on the portion of the virtual display used by the ANSI terminal emulator. Upon return the top half of D0 will contain the index of the terminal character row which contains that point. The lower half of D0 will contain the index of the character column.

Display Function 32: RCtoXY

Convert terminal row and column to screen coordinates. The top half of D1 contains the index of a terminal character row and the lower half of D1 contains the index of the character column. Upon return D0 will contain the coordinate of the upper left corner of the character cell. The top half of D1 will contain the width (in pixels) of the character cell and the bottom half will contain the height of the character cell.

KEYBOARD AND MOUSE EVENT PROCESSING

The event manager creates a buffered stream of 16-bit values which encode actual events. In general, the high-order 4 bits of the values are event type codes and the low-order 12 bits are event parameters. The following event-type codes are assigned:

0 delta time
1 mouse X location
2 mouse Y location
3 key or button pressed
4 key or button released
5 absolute time

Whenever the keyboard or mouse changes state, a time event is generated (either a type 0 or type 5 event) which reports the time of the event. This is followed by an event value which specifies the actual change which occurred.

EVENT MANAGER FUNCTIONS

The operating system provides access to the event manager functions through the same mechanism as to the display functions. The trap #13 instruction invokes the function whose code is passed in register D0. The event manager functions are described in the following paragraphs.

Event Function 40: eventsEnable

Turns on the event manager. Any subsequent user input action will cause event values to be created. Normal keyboard input through the "console" device and terminal emulator is disabled.

Event Function 41: eventsDisable

Turns off the event manager. Keyboard input through the "console" device and terminal emulator is enabled.

Event Function 42: eventSignalOn

Requests the event manager to signal the current process when events occur. The event signal is disabled after being issued.

Event Function 43: eventMouseInterval

Specifies how frequently mouse motion events are to be created if the mouse is continuously moving. The frequency value is passed in register D0 and is specified in units of milliseconds granularity of milliseconds). A value of 0 indicates that mouse motion events should not be created

Event Function 44: getEventCount

Returns in register D0 the number of event values in the event buffer waiting to be processed.

Event Function 45: getNewEventCount

Returns in register D0 the number of event values in the event buffer which have occurred since the previous call to this function.

Event Function 46: getNextEvent

Returns in register D0 the next event value in the event buffer.

Event Function 47: getMillisecondTime

Returns in register D0 the number of milliseconds since the system was turned on (a 32-bit value).

Event Function 48: setAlarmTime

A 32-bit millisecond time relative to system power-up is passed in register D0. The requesting process will be signaled when this time is reached.

Event Function 49: clearAlarm

Clears any pending alarms that the process has requested.

EVENT MANAGER KEY CODES

Each key on the keyboard, each position of the joydisk, and each of the mouse buttons has an event driver code associated with it. Table 8-1 shows the event code associated with each button.

Key Label	Event Code	Key Label	Event Code
Caps lock	139	S	115
Shift (left)	136	T	116
Shift (right)	137	U	117
Control	138	V	118
	140	W	119
Break	141	X	120
Backspace	8	Y	121
Tab	9	Z	122
Line Feed	10	[{	91
Return	13	\`	92
Escape	27	]}	93
(space bar)	32	|~	124
'"	39	Rubout	127
> .	46	Enter	150
< ,	44	Pad ,	151
- _	45	Pad -	152
/ ?	47	Pad .	153
0)	48	Pad 0	154
1 !	49	Pad 1	155
2 @	50	Pad 2	156
3 #	51	Pad 3	157
4 $	52	Pad 4	158

5 %	53		Pad 5	159	
6 ^	54		Pad 6	160	
7 &	55		Pad 7	161	
8 *	56		Pad 8	162	
9 (	57		Pad 9	163	
; :	59		F1	201	
= +	61		F2	202	
A	97		F3	203	
B	98		F4	204	
C	99		F5	205	
D	100		F6	206	
E	101		F7	207	
F	102		F8	208	
G	103		F9	209	
H	104		F10	210	
I	105		F11	211	
J	106		F12	212	
K	107		Mouse left	130	
L	108		Mouse middle	129	
M	109		Mouse right	128	
N	110		Joydisk up	213	
O	111		Joydisk down	214	
P	112		Joydisk right	215	
Q	113		Joydisk left	216	
R	114				

FLOATING POINT SUPPORT

The operating system provides access to the floating point hardware. Floating point values are in IEEE format. Both 32-bit single precision and 64-bit double precision formats are supported.

These operations are invoked by a trap #12 instruction with function code and arguments stored in registers. The floating point function code is passed in register D2. Operands are passed in registers D0 and A0 if they are single precision or integer, or in register pairs D0/D1 and A0/A1 if they are double precision. If only one operand is required it is passed in D0 (or D0/D1). The result is returned in register D0 for single precision, and in register pair D0/D1 for double precision.

For subtracts, compares, and divides the value in register A0 (or A0/A1) is subtracted from, compared to, and divided into the value in register D0 (or D0/D1). For compare operations the processor's condition codes are set to reflect the result of the comparison. The floor function converts a floating point number to the largest integer less than or equal to it. The file "/lib/sysfloat" contains symbolic definitions of the floating point functions for use by assembly language programs.

FLOATING POINT FUNCTIONS

FP Function 0: FADD

Add two single precision numbers.

FP Function 1: FSUB

Subtract two single precision numbers.

FP Function 2: FMUL

Multiply two single precision numbers.

FP Function 3: FDIV

Divide two single precision numbers.

FP Function 4: FCMP

Compare two single precision numbers.

FP Function 5: FNEG

Negate a single precision number.

FP Function 6: FABS

Take absolute value of a single precision number.

FP Function 7: FItoF

Convert integer to single precision floating point.

FP Function 8: FFtoIr

Round single precision floating point to integer.

FP Function 9: FTtoIt

Truncate single precision floating point to integer.

FP Function 10: FFtoIt

Floor function for single precision numbers.

FP Function 11: FFtoD

Convert single precision number to double precision.

FP Function 12: FDtoF

Convert double precision number to single precision.

FP Function 13: FDADD

Add two double precision numbers.

FP Function 14: FDSUB

Subtract two double precision numbers.

FP Function 15: FDMUL

Multiply two double precision numbers.

FP Function 16: FDDIV

Divide two double precision numbers.

FP Function 17: FDCMP

Compare two double precision numbers.

FP Function 18: FDNEG

Negate a double precision number.

FP Function 19: FDABS

Take absolute value of a double precision number.

FP Function 20: FItoD

Convert an integer to double precision floating point.

FP Function 21: FDtoIr

Round double precision floating point to integer.

FP Function 22: FDtoIt

Truncate double precision floating point to integer.

FP Function 23: FDtoIt

Floor function for double precision numbers.

FP function 24: FsetStat

The value in D0 is written into the 32081's Status Register. Bits 7 and 8 may be used to specify a rounding mode. Bits 9-15 may be used to store an arbitrary value. No other bits have any effect if set. Note that changing the rounding modes will have a global effect on all processes using the floating point processor.

FP Function 25: FgetStat

The value of the 32081's status register is returned in D0.

FLOATING POINT RETURNS

A successfull execution of a floating point system call returns with the V (overflow) bit cleared. If an error occurs, the routine returns with the V bit set and the error indicated by the contents of Register D0. The error codes are:

$8000	FPU failed to complete an operation.
$4000	Driver called with invalid operand (>25) in D2.
$0001	Result had an underflow. "Trap on underflow" was enabled.
$0002	Result overflowed.
$0003	Divide by zero error.
$0004	Invaled op operand passed to FPU. (This error should not ever occurr.)
$0005	FPU passed an operand that is not a valid floating point value.
$0006	Result was inexact with "trap on inexact result" enabled.

MEMORY UTILIZATION

OVERALL ADDRESS SPACE

The 68010 processor on the 4404 is capable of addressing 16 Mb of memory. Of this, the 4404 recognizes the lower 8 Mb. (All addresses given in this discussion will be hexadecimal unless stated otherwise.) The 4404 operating system uses a virtual memory scheme whereby 8 Mb of virtual memory is mapped into the 4404's physical memory in 4 Kb increments. To a programmer working through the operating system, it appears that the entire 8 Mb address space (ranging from 000000 through 1FFFFF) is available.

PHYSICAL MEMORY

The standard 4404 contains 1 Mb of physical RAM in addresses 000000 through 0FFFFF. Option 1 adds an additional 1Mb of physical memory in addresses 100000 through 2FFFFF.

Addresses 200000 through 5FFFFF are reserved for future expansion.

DISPLAY MEMORY

The 4404 display memory begins at address 600000 and occupies through address 6FFFFF. The virtual display begins in the upper left corner at address 600000 and proceeds in 1024 (decimal) lines of 64 (decimal) 16-bit (decimal) words. Each word has the most significant bit first, thus each word controls 16 pixels on the display.

I/O AND ROM MEMORY SPACE

The memory segment from 700000 through 7FFFFF is dedicated to ROM, I/O, and various utilities. It consists of eight 128 Kb pages arranged as:

700000 -- 71FFFF	Spare 0
720000 -- 73FFFF	Spare 1
740000 -- 75FFFF	Boot ROM
760000 -- 77FFFF	Debug ROM space (for factory use)
780000 -- 79FFFF	Processor Board I/O (treated later)
7A0000 -- 7BFFFF	Peripheral Board I/O (treated later)
7C0000 -- 7FFFFF	EPROM application space

Processor Board I/O

780000 -- 781FFF	Map Control Registers
782000 -- 783FFF	Video Address Registers
784000 -- 785FFF	Video Control Registers
786000 -- 787FFF	Spare
788000 -- 789FFF	Sound
78A000 -- 78BFFF	Floating Point Hardware
78C000 -- 78DFFF	Debug ACIA
78E000 -- 78FFFF	Spare

Peripheral Board I/O

7A0000 -- 7AFFFF	Reserved for future expansion
7B1000 -- 7B1FFF	Diagnostic registers
7B2000 -- 7B3FFF	Printer
7B4000 -- 7B5FFF	Serial I/O
7B6000 -- 7B7FFF	Mouse
7B8000 -- 7B9FFF	Timer
7BA000 -- 7BBFFF	Calendar
7BC000 -- 7BDFFF	SCSI bus address registers
7BE000 -- 7BFFFF	SCSI

Section 9

"EDIT" THE TEXT EDITOR

INTRODUCTION

This section describes "edit," the standard 4404 text editor, including how to call the editor, the interface between the editor and the 4404 operating system, a description of each of the editor commands (with examples), and an annotated list of the messages that the editor may issue.

"edit" is both content-oriented and line-oriented. Lines in the file being edited may be referenced either by specifying a line number or by specifying some part of the content of the line. "edit" is not a screen-oriented editor.

CALLING THE EDITOR

Example:

```
edit
```

When the editor is called with no arguments, it issues a message that a new file is being created, and then prompts for the information that is to be put into the file. When the editing session is terminated (by the "stop" command, for example), the editor will prompt for the name of the file to which to write the information. The user responds to this prompt by typing in the file name, including a path name if necessary.

If an end-of-file signal is typed in response to the prompt for a file name, all information is discarded and the editing session is terminated. (See the discussion "Operating System Interface" later in this section for more information on the end-of-file signal.)

Calling the Editor with a File Name

Example:

```
edit test
```

If only one file name is given as an argument, the editor assumes that this is the file or the name of the file that is being edited.

If the file does not exist, a new file having the specified name is created. A message stating that fact is issued, and the editor then prompts for the information to be stored in the file. When the editing session is terminated, the information is written to the file.

If the file already exists, the information in it is read into an edit buffer and a prompt for an editor command is issued. When the editing session is terminated, the file will contain the revised information. The information as it was before the editor was called is preserved in a backup file (unless the "b" option was specified, as described later on). The name of the backup file is normally the name of the original file with the characters ".bak" appended to the end of it. If the original name is too long to accommodate the additional four characters, the name is truncated and the ".bak" appended to the shortened name.

Calling the Editor with Two File Names

Example:

```
edit test newtest
```

When the editor is called with two file names, the first file name is assumed to be the name of the file containing the information to be edited, and the second name is that of the file that is to receive the revised information. Both file names may contain path names if necessary to adequately describe their locations. If a path name is specified for the first file name, it is not propagated to the second file name. In the example, the file "test" is assumed to contain the information which is to be edited, and the file "newtest" is going to contain the edited information. If the first file does not exist, the editor writes a message indicating that the edit file does not exist, and then terminates the edit session. If the second file already exists, a prompt is issued asking for permission to delete the existing file. (This prompt may be avoided with the "y" option, described below.) If an end-of-file signal is typed in response to this prompt, it is assumed that the file is not to be deleted, and the editing session is immediately terminated with no changes having been made.

Options

Options are specified to the editor by specifying an argument whose first character is a plus sign (+). The plus sign is immediately followed by one or more lowercase letters indicating the option or options selected. The options may be before, after, or intermixed with file name arguments.

b Do not create a backup file. "b" tells the editor not to create a backup file.

n Do not initially read the file being edited. This option is meaningful only if an existing file is being edited. Normally, the editor reads the file into memory so that the information may be manipulated with editor directives. By specifying "n" as an option, the information is not initially read into memory. The user may then use editor directives to enter new information, either from the terminal or by reading other files, which will appear in front of the information in the file being edited. The "new" command must be used to start the reading of the edit file. This option is most useful if a large amount of information is to be entered in front of the data being read from the file being edited. To insert only a small amount of information at the front of a file, the "insert" command may be used.

y Delete any existing copy of the new file or the backup file. "y" causes the editor to delete any existing copy of the backup file (if only one file name is specified) or the new file (if two file names are specified), without asking permission from the user.

If the editor cannot recognize an argument as a valid option, it issues an error message and continues to look for valid arguments.

Examples of calls including options:

```
edit test +b
edit test newtest +y
edit +nb test
```

OPERATING SYSTEM INTERFACE

The text editor follows the operating system conventions with regard to special characters and file names. For a discussion of file names, see Section 1 of this manual. The special characters and their effect on the editor are treated below.

Normally, the editor allows any character to be in a file, including control characters. There are some characters, however, which have special meaning to the operating system and thus cannot be typed in from the keyboard. The special characters with which the editor is concerned are:

- backspace character
- escape character
- line delete character
- horizontal tab character (control-i)
- control-d: keyboard signal for end-of-file
- control-c: keyboard interrupt
- control-\ : "quit" signal

Backspace character

The backspace character (Back Space on the keyboard) is used when entering commands and data to erase the last character typed.

Escape character

The ASCII "escape" character (Esc on the 4404 keyboard) is used to temporarily stop and resuming the printing of information at the terminal. A more detailed description of the function of the "escape" character is described in the documentation of the 4404 Operating System. Here, it suffices to say that it is not possible to enter the "escape" character into a file using the editor.

Line delete character

The line delete character is used when entering commands and data to delete the line currently being typed.

Horizontal tab character

This character (Tab from the 4404 keyboard) refers to the ASCII horizontal tab character (HT), a hexadecimal 09. This is not the same as the tab character that can be defined within the editor. The editor itself is not concerned with the HT character, but the operating system may perform special handling when this character is typed or displayed. The editor treats the HT character as a single character, regardless of how the 4404 displays it.

Control-d: keyboard signal for end-of-file

The editor treats a "control-d" as an "end-of-file". The action taken by the editor depends on what the editor was expecting as input. A "control-d" typed in the middle of a command has the same effect as a line delete character. If the control-d is typed as the first character in response to a request for a command (that is, in response to the # prompt), it is treated as a "stop" command. A "control-d" typed while inserting lines has the same effect as typing the line delete character followed by the line number character and a carriage return. That is, it cancels the current input line and the editor requests an editor command.

The effect of typing control-d in response to specific prompts depends on the prompt that was issued. Each such case is treated in the "Editor Command" discussions.

Control-c: keyboard interrupt

The editor traps the "control-c" keyboard interrupt and uses it as a signal to stop executing an "append", "cchange", "change", "find", or "print" command. It has no effect on other commands. If the editor is executing multiple commands typed on a single line, typing a "control-c" will cause the editor to stop processing those commands and request a command from the keyboard.

Control-\ : "quit" signal

The "quit" signal causes the editor to terminate immediately, without making any attempt to save the edited information. If an existing file was being edited when the "quit" signal was typed, the original file is left intact without any of the changes that had been made during the edit session.

THE EDITOR'S USE OF DISK FILES

The standard 4404 text editor is a disk-oriented editor: the information being edited is read from and written to disk files. Other than the user's terminal, the only way to provide information to the editor is through disk files. When the editor is called to edit an existing file, the information in that file is read into a large buffer in memory called the "edit buffer". It is in this buffer that all of the changes to the information take place. When the user is satisfied with the changes made, the updated information is written to a disk file in response to specific commands. If a file is larger than will fit in the edit buffer, the file must be processed in segments. With few exceptions, the editing commands operate only on data that is in the edit buffer. Commands are provided which permit the user to flush the edit buffer of updated information and read in the next segment of data for editing. How the editor manipulates disk files depends on whether it is creating a new file or editing an existing file. In some cases, a temporary file is created to hold the updated information. If used, this temporary file is named "edit" followed by a period, 5 digits, and a single letter; for example, "edit.00324a." Unless the editor is terminated by a "quit" signal or a fatal system error, the temporary file is destroyed at the end of the edit session.

CREATING A NEW FILE

When the editor is called with a single file name and that file does not already exist, the editor will create the file at the start of the edit session and write directly into it as the edit session progresses.

When the editor is called with no file names specified, a temporary file in the user's current directory is created and the information is written to it as the edit session progresses.

At the end of the edit session, this temporary file is given the name specified in response to the "File name?" prompt.

EDITING AN EXISTING FILE

When the editor is called with a single file name, and that file already exists, a temporary file is created and the information is written to it as the edit session progresses. The temporary file is created in the same directory in which the file being edited resides. At the end of the edit session, the original file is renamed to the backup file name, and the temporary file is given the name of the original file. If no backup file is requested (by specifying a "b" option), the original file is destroyed and the temporary file is given the name of the original file.

When the editor is called with two file names specified, the second file is created and the updated information is written directly into it. The original file is not changed.

COMMAND INPUT FROM A FILE

It is possible to use I/O redirection to have the editor read its commands from a file instead of from the keyboard. The editor will process the commands as though they were entered from the terminal's keyboard. If the end of the command file is reached before a "stop" or "abort" command is read, the action is the same as though a "control-d" were typed from the keyboard. (See the discussion of "control-d earlier in this section.)

FATAL ERRORS

The text editor attempts to make an intelligent decision when confronted with an error response to an operating system call. However, if an error is received which is unexpected and indicates that the editor cannot continue to function, it will issue a message and terminate immediately. The various messages, both fatal and nonfatal, are listed under the heading "Editor Messages" later in this section.

EDITOR COMMANDS

USING STRINGS

Several editor commands use character strings as arguments. These arguments are either matched against strings in the text, or replace a string in the text. A string argument begins after a delimiter character and continues as a sequence of any characters until the delimiter is again encountered. The delimiters are not considered part of the string to be used in the matching or replacement operations. Although the delimiters in the following descriptions are frequently represented as slashes, "/", nearly any non-blank, non-alphanumeric character may be used as the delimiter such as: * / ( ) $, . [ ] : ' etc. Note that the following characters may not be used to enclose strings unless they are preceded by either a plus (+) or minus (-) sign: "^" (denotes first line of file), "!" (denotes last line of file), "-" (denotes target is above current line), and the character denoted by "lino" (normally a pound sign), which is used to indicate line numbers. The equals sign "=" may not be used as a string delimiter. The delimiter character is redefined in each new request by its appearance before a string. If two strings exist in one command (as in the "change" command), the same delimiter character must be used for each string.

All editor commands use the <line> information preceding the command to position the pointer prior to any command action. The <line> parameter may of course be null, meaning leave the pointer at its current position. All of the following are valid <line> designators:

Any number	The specific line number
+n	The nth subsequent line
n	The nth previous line
/<string>/	The next line in the file containing the indicated string of characters
-/<string>/	A previous line containing the indicated string
^	The first line of the file
!	The last line of the file
null	The current line

Line numbers less than 1.00 must be specified with a leading zero. For example, even though the editor may display a line number as ".10", it should be specified as "0.10" when used in commands. The maximum line number is 65535.99. Inserting after this maximum line number will cause the line numbers to "wrap around" back to zero.

Many editor commands require <target> information. This tells the editor to operate on the "current" line and all other lines in the file up to the line referenced by the <target>. In cases where a <target> is required, leaving it null will make the <target> default to one, and only the current line will be affected. All of the following are valid <target> designators:

an integer n	n lines should be affected by the edit operation
#n	The line number of the last line to be affected. The "#" is actually the "lino" character and may be changed by the user with the "set" command.
/<string>/	The next line in the file containing the specified character string.
-/<string>/	The previous line containing the indicated string
^	All lines up to the top of the file
!	All lines to the bottom or last line of the file
+or- n	Indicates that n lines should be affected and in which direction from the current line
(null)	Defaults to 1 and only the current line is affected

As we have seen, <target> is used to specify a range of lines to which the command will apply. The command will be applied to each line, starting with the line specified by <line> and continuing until the target is reached.

If a string <target> is specified, the command will apply to successive lines of text until a line containing the string is reached. Processing proceeds downward in the edit buffer unless the target is preceded by a "-" (minus sign), indicating that processing is to proceed upward (toward the first line) in the edit buffer. Targets may also be preceded by a plus sign (indicating downward movement). If a line number target is specified, processing begins at <line> and proceeds toward the target line number. Some examples of <target>s are:

```
2
+10
-3
/STRING/
+/STRING TARGET/
-/BACKWARD DISPLACEMENT TO A STRING/
+*ANY DELIMITER WILL WORK FOR STRING*
++EVEN PLUS SIGNS CAN WORK+
#23.00
```

SPECIFYING A COLUMN NUMBER

Any "/<string>/" descriptor may be postfixed with a column number immediately after the second delimiter to indicate that the preceding string must begin in the column specified. If the column specified is not in the range of the zone in effect, the request will be ignored. (See the "zone" command.) Some examples are:

```
/IDENT/11
/PROGRAM/77
*LABEL*2
$COMMENT$30
```

USING THE DON'T-CARE CHARACTER

A "Don't-Care Character" may be set to allow indiscriminate matches of parts of a string. When this character is placed in a string, any character in the file will automatically match. The Don't-Care Character will have its special meaning only in a string being used to search the file. In other words, the Don't-Care Character will not act as such in a replacement string such as the second string of a "change" command. The Don't-Care Character may be effectively disabled by setting it to a null. Assuming we have previously set the Don't-Care Character to a "?", here are some examples:

/A???/	Matches any 4-letter string beginning with A
@03/??/78@	Matches all days in the 3rd month of 1978
/???/9	Matches any 3-letter string starting in column 9

THE COMMAND REPEAT CHARACTER:

The "command repeat character," control-r, repeats the last command in the input buffer. Some examples of commands which may be useful to repeat are:

PRINT 15	To print a screen of lines at a time
NEXT	Allows you to single step through the file with one key
^CO!!	To quickly fill the workspace
FIND/SOME STRING/	If the first string found is not the one desired

USING THE EOL CHARACTER

The editor supports an "eol" or "End Of Line" character to allow multiple commands in a single line. There are some commands that cannot be followed by another command on the same line. This fact is documented in the descriptions of those commands. The "eol" character may be changed by using the editor's "set" command. An example of "eol" use (with "eol" set to "$") is:

```
^D2$P10$T
```

This sequence will delete the first 2 lines of the file, then print the next 10 lines, and finally return the pointer to the top of the file.

USING TABS:

You may specify a tab character and up to 20 tab stops. The tab character may then be inserted into a line, where it will be replaced by the appropriate number of fill characters when the end of the line is received. The fill character defaults to a space, but may be changed to another character with the editor's "set" command. If tab stops or the tab character have not been previously set, but some character has been used throughout the file as a tab, it can still be expanded by setting it to be the tab character, setting up your tab stops and then using the "expand" command on the file.

Note that if the tab character has been set, subsequent uses of the "insert" or "replace" commands will cause automatic tab expansion. However if a tab character is added to the file by the use of a "change", "append", or "overlay" command, that character will remain intact in the file until the "expand" command is invoked on the line containing that tab character.

After tabs are expanded, the tab character no longer exists in the data. All occurrences will have been replaced by the appropriate number of fill characters. Setting the tab character to be the same as the fill character effectively disables the tab feature. Note the the tab character described above is distinct from the ASCII horizontal tab character (HT or control-i). The effect of the HT character is described in the "Operating System Interface" discussion earlier in this section. It is possible to set the editor tab character to the HT character. If this is done, the operating system may take special action when the HT character is typed, but the character will be replaced by fill characters when it is put into the edit buffer.

LENGTH OF TEXT LINES

Lines entered from the keyboard are limited to 255 characters. The lines in the text file may be of any length. Lines longer than 255 characters may be created with the "merge" and "append" commands.

COMMANDS

There are five groups of editor commands: environment commands, system commands, "current line" movers, edit commands, and disk commands. A complete description of all commands in each group is given below. In the following descriptions, quantities enclosed in square brackets ([...]) are optional and may be omitted. A backslash ( ) is used to separate options. Many commands have abbreviations. Both the full name of the command and its abbreviation are given. A command and its abbreviation may be used interchangeably. All commands below are in lower case; however, in use, a command may be in either upper case or lower case.

ENVIRONMENT COMMANDS

DK1

Syntax

dk1 <command string>

Description

"dk1" is used to define one of two "command constants", which can be executed at any time by the "kl" command. The <command string> is a single command or several commands separated by the "eol" character (see "set" command). All of the command line, including the carriage return is assumed to be the argument to the "dk1" command. The "dk1" command is most useful for remembering and re-executing a frequently used sequence of commands.

Example

dk1 f -/.nl/1$i/.sp	Define a command sequence of "f -/.nl/1" followed by "i/.sp". This assumes that "eol" is "$". This sequence may be executed by typing "k1".

DK2

Syntax

dk2 <command string>

Description

"dk2" is used to define one of two "command constants", which can be executed at any time the the "k2" command. The <command string> is a single command or several commands separated by the "eol" character (see "set" command). All of the command line, including the carriage return is assumed to be the argument to the "dk2" command. The "dk2" command is most useful for remembering and re-executing a frequently used sequence of commands.

Example

dk2 c /sample// 1 2	Define the command constant: "c /sample// 1 2". This command may be executed by typing "k2".

ESAVE

Syntax

esave [<path_name>]

Description

The "esave" command saves the current editor "environment" on an "editor configuration" disk file named ".editconfigure" in the user's directory. The editor environment consists of the "header" column count; the "numbers" and "verify" flags; current tab stops; the "tab", "dcc", "fill", "eol", and "lino" characters; the commands saved as command constants "k1" and "k2"; and the search zones in effect. When the editor is called, the environment is automatically set from the configuration file in the user's directory, if one exists. The editor environment may also be reset from the configuration file at any time during the edit session by the "eset" command, described below.

The environment information may be saved in a directory other than the user's current directory by specifying a path name as an argument to the "esave" command. This path must include only directory names and must be terminated by the pathname separator "/".

Examples

esave	Save the current editor environment on the file ".editconfigure" in the user's directory.
esave /dde/	Save the current editor environment in file "/dde/.editconfigure".

ESET

Syntax

eset [<path_name>]

Description

The "eset" command is used to reset the editor environment from an editor "configuration" file created by the "esave" command (see above). The configuration file is named ".editconfigure" and is normally expected to be found in the user's current directory. A path name may be specified as an argument to the "eset" command to force the searching of a different directory. This path must include only directory names and must be terminated by the pathname separator "/".

Examples

eset	Reset the editor environment from the file ".editconfigure" in the user's directory.
eset /dde/	Reset the editor environment from file "/dde/.editconfigure".

HEADER

Syntax

header [<count>]
h [<count>]

Description

A header line of <count> columns will be displayed. The heading consists of a line showing the column numbers by tens, followed by a line of the form "123456789012..." to indicate the column number. Columns for which tab stops are set will contain a hyphen instead of the normal digit. If a column count is given, it becomes the default so that if just "h" is subsequently typed, that number of columns will be printed.

Examples

header 72	Display column number headings for 72 columns
h 30	Display column numbers for 30 columns

K1

Syntax

k1

Description

Execute the command constant that was defined by "dk1". If no command constant was defined, the current line is printed. This command may not be followed by another command on the same line.

Examples

k1 Execute the command constant.

K2

Syntax

k2

Description

Execute the command constant that was defined by "dk2". If no command constant was defined, the current line is printed. This command may not be followed by another command on the same line.

Example

k2 Execute the command constant.

LK1

Syntax

lk1

Description

Display the command constant that was defined by "dk1". If no command constant was defined, a blank line is printed.

Example

lk1 Display the command constant.

LK2

Syntax

lk2

Description

Display the command constant that was defined by "dk2". If no command constant was defined, a blank line is printed.

Example

```
lk2  Display the command constant.
```

NUMBERS

Syntax

numbers [off/on]
nu [off/on]

Description

The line number flag is turned off or on. If the flag is off, then line numbers will never be printed. If neither "off" nor "on" is specified, then the flag will be toggled from its current state.

Examples

```
numbers off    Turn line number printing off

nu on      Turn it back on

nu   Toggle from on to off or from off to on
```

RENUMBER

Syntax

renumber
ren

Description

The "renumber" command will renumber all of the lines in the current edit buffer. Lines in the renumbered buffer will start with the line number of the first line in the buffer and will have an increment of one. The current line does not change, although its number will probably have been changed.

Examples

renumber Renumber the lines in the current edit buffer

ren Renumber the lines in the current edit buffer

SET

Syntax

set <name> = '<char>'

Description

set" is used to define certain special characters or symbols. The <name>s which may be set are:

tab	the tab character
fill	the tab fill character
dcc	the "don't care" character for string searches
eol	the end of line character which may be used to separate several commands on a single line
lino	the line number flag character which is used to indicate that a target is a specific line number

The default values are: dcc, tab, and eol are null, fill is the space character, lino is "#"

The default values may be initialized from a configuration file in the user's directory. See the "esave" command.

Examples

`set tab='/'`	Set the tab character to a slash
`set tab=''`	Disable tabbing by setting the tab character to a null
`set fill=' '`	Set tab fill character to a blank
`set eol='$'`	Set the EOL character to $
`set lino='@'`	Set the line number flag to @

TAB

Syntax

tab [<columns>]

Description

Used to set the tab stops. All previous tab stops are cleared. If no columns are specified, then the only action is to clear all tab settings. Any tab characters occurring beyond the last tab stop are left in the text. The maximum number of tab stops allowed is 20. Tab stops MUST be entered in ascending order.

Examples

`tab 11,18,30`	Set tab stops at columns 11, 18, and 30
`tab`	Clear all tab stops

VERIFY

Syntax

verify [on/off]
v [on/off]

Description

The verify flag is turned on or off. The verify flag is used by the commands "change" and "find" (and several others) to display their results. If neither "on" nor "off" is specified, then the flag will be toggled from its current state.

Examples

`verify off`	Turn verification off
`v on`	Turn it back on

ZONE

Syntax

zone [c1,c2]
z [c1,c2]

Description

zone" is used to restrict all sub-string searches (find, change, <target>s, etc.) to columns "c1" through "c2" inclusive. Any substrings beginning outside those columns will not be detected. If "c1" and "c2" are not specified, then the zones will be reset to their default values (columns 1 and 255). A string which starts within the specified search zone and extends out of it will still match a target.

Examples

zone 11,29 Restrict searches to columns 11 through 29

zone Search columns 1 through 255

SYSTEM COMMANDS

ABORT

Syntax

abort

Description

This command terminates the edit session without saving any of the changes made during that session. The original file, if one exists, is left intact. When typed, this command will prompt "Are you sure?". If a "y" is then typed, the edit session will be terminated. Typing an "n" or end-of-file signal will cause the editor to look for another command. Typing any other character will cause the prompt to be issued again.

Examples

abort	Abort the editing session.

EDIT

Syntax

edit <editor arguments>
e <editor arguments>

Description

The "edit" command causes the current editing session to be terminated (as though a "stop" or "log" command had been entered), and another editing session started. The <editor arguments> are any valid file names and editor options as described earlier in this section under the heading "Calling the Editor". This command may not be followed by another command on the same command line. All changes to the editing environment made by "Environment Commands" remain in effect.

Example

edit test +b	Terminate the current editing session and start editing file "test" with editor option "b".

LOG

Syntax

log

Description

This command ends the editing session. The updated information is written to the new file, and, if necessary, any unprocessed data from any existing file is copied to the new file. A backup file is created if circumstances warrant it. (see the "Operating System Interface" discussion earlier in this section for more information on the editor's handling of disk files at the end of an editing session.)

Example

```
log
```

STOP

Syntax

stop
s

Description

Same as "log".

Examples

stop

s

U

Syntax

u <operating_system_command>

Description

The "u" command permits the execution of an operating system command. The specified command is passed to the "shell" program for execution. The editor waits for the operating system command to finish before prompting for another editor command. This command may not be followed by another editor command on the same line.

Examples

```
u list test     List the file "test"

u copy test test1   Copy the file "test" to "test1"
```

WAIT

Syntax

wait

Description

The "wait" command is used to wait for the completion of a background task generated by the "x" command (described below). This command cannot be used to wait for completion of a background task that was not generated by the editor. The editor will not request a command until the background task is completed or a keyboard interrupt (control-c) is typed. When the background task terminates, a message is displayed specifying the task number and whether it completed normally or abnormally. In the event of abnormal termination, the response code or interrupt code that caused the termination is given.

Example

```
wait      Wait for the background task to complete
```

X

Syntax

x <operating_system_command>

Description

The "x" command is used to start a background task running. The <operating_system_command> which was specified as the argument is passed to the "shell" program for execution. The task generated must run to completion before the editor will allow the generating of another such background task. The "wait" command must be used to receive the termination status of a task before the "x" command may be used again. This command may not be followed by another command on the same line.

Example

x copy test test1	Copy "test" to "test1" as a background task. A "wait" command must be used to determine the termination status of the task before another background task can be generated.

"CURRENT LINE" MOVERS

BOTTOM

Syntax

bottom b

Description

Moves to the last line in the file and makes it the current line.

Examples

bottom Make the last line of the file the current line

b Make the last line of the file the current line

FIND

Syntax

find <target> [<occurrence>]
f <target> [<occurrence>]

Description

Moves the current line pointer to the line specified by <target> and makes it the current line. If the verify flag is on (see "verify"), the line will be printed. If <occurrence> is specified (an unsigned integer or an asterisk), the command will be repeated <occurrence> times. If <occurrence> is an integer, it must not start in the first column following the second delimiter of a string <target>, as it would then appear to be a column specifier for that string. If no column is to be specified, insert a space after the second delimiter and before the <occurrence>, as in the second example given below. An asterisk means all occurrences of the <target> will be found until the bottom or top of the edit buffer is reached. If the target is not found, the current line pointer will not be moved.

Examples

find /string/ Find the next line containing the string "string"

f/three lines/ 3 Find the next three lines containing the string "three lines"

```
f/all 'til bottom/*        Find all following occurrences of
                           the indicated string

f-/program/7 *        Find all previous lines which have the
                      word "program" starting in column seven
```

NEXT

Syntax

```
next [<target> [<occurrence>]]
n [<target> [<occurrence>]]
```

Description

The line specified by the target is made the current line. If the verify flag is on (see "verify"), the line will be printed. If <occurrence> is specified, it must be an unsigned integer. It indicates which occurrence of a line containing the target is to be made the current line. If the target is not reached, the current line pointer will be positioned at the bottom of the edit buffer (or top of the edit buffer for a negative <target>). If no target is specified, the next line will be made the current line.

Examples

```
next 5     Make the fifth following line the current line

n     Make the next line the current line

n-10       Make the 10th previous line current

n/string target/     Make the next line containing "string
                     target" to be the current line

n/3rd occurrence/3         Make the third line containing the
                           indicated string the current line
```

POSITION

Syntax

position <target>
pos <target>

Description

Searches forward through the file for an occurrence of <target> and makes the line in which it occurs the current line. If the target is not found in the current edit buffer, the edit buffer is flushed and the next edit buffer is read from the file being edited. This process continues until the target is located or the end of the file is detected. If the target cannot be located, the current position is the first line in the last edit buffer.

The <target> may not be a "backwards target" (preceded by a minus sign) and may not be an integer indicating relative displacement. Only a string or a line number (preceded by the "lino" character) are valid targets. Search zones are honored during the search for the target. A column number is allowed after the target, but an occurrence specification is not permitted.

Examples

position /string/5	Position to the line containing the string "string" in column 5.
pos #1000	Position to line number 1000

TOP

Syntax

top
t

Description

The first line of the file becomes the current line.

Examples

top	Make the first line of the file the current line

EDITING COMMANDS

APPEND

Syntax

append /<string>/ [<target>]
a /<string>/ [<target>]

Description

Appends the specified <string> after the last character of the current line (and to successive lines until the target is reached).

If the string is postfixed with a column number, then the string is added beginning at the specified column (rather than at the end of the line). Any characters previously in the line following the specified column are overwritten.

Examples

append /./	Append a period to the end of the current line
a *HELLO*	Append the word "HELLO" to the end of the current line and to the end of the next line.
a/sequence/73 *END*7	Append the word "sequence" starting in column 73 of the current line and successive lines until a line containing the characters "END" beginning in column seven is found.

BREAK

Syntax

break

Description

The "break" command allows the splitting of a line into two lines. The current line is printed, then a line of input is accepted from the terminal (the break line). When the line is printed, all ASCII HT characters will be displayed as spaces so that the terminal cursor will not be artificially advanced. The break line will be positioned directly beneath the line printed out.

In response to the "Break---" prompt, type any characters to move the cursor until it is beneath the character that is to be the first character of the second line. Then type a carriage return.

After the line is split, the second half of the broken line becomes the current line. If you type an end-of-file signal in response to the "Break---" prompt, the current line will not be changed. The current line will also not be changed if the carriage return typed in the break line is beyond the end of the current line.

Example

```
break
     25.00 This is the current line.
   Break---xxxxxxxxxxxx
```

The line will be broken at the start of the word "current".

CHANGE

Syntax

change /<string1>/<string2>/ [<target> [<occurrence>]]
c /<string1>/<string2>/ [<target> [<occurrence>]]

Description

Replaces <string1> with <string2>. If <string2> is omitted, <string1> is deleted. If no <target> is specified, only the current line is affected. The slashes represent any non-blank delimiter character.

<occurrence> specifies which occurrence of <string1> is to be replaced in each line. It is either an unsigned integer or an asterisk (*) signifying that all occurrences of the substring <string1> are to be replaced with <string2>. By default, only the first occurrence will be changed. Note that if <occurrence> is specified, and if changes are to occur to the current line only, then the target should be "1."

Examples

```
change /this/that/   Replace the first occurrence of "this"
                     in the current line with "that"

c/A/B/ 1*            Change all occurrences of "A" in the current
                     line to "B"
```

c /first/last/10	Change the first occurrence of "first" to "last" in the current line and also in the nine following lines
c /new/old/ /a target/	Change the first occurrence of "new" to "old" in each line down through the line containing the string "a target"
c ,a,, -10 *	Remove all "a"s in the current line and in the nine preceding lines
c*Hello*	Delete the character string "Hello" from the current line

CCHANGE

Syntax

cchange /<string1>/<string2>/ [<target> [<occurrence>]]
cc /<string1>/<string2>/ [<target> [<occurrence>]]

Description

cchange" stands for Controlled Change. This command is exactly like the normal "change" command except that you can interactively specify whether each line containing <string1> should actually be changed or left as is. This allows you to step through the edit buffer and selectively change certain strings. When a line containing <string1> is found, it is displayed at the terminal and you receive a prompt, "Change?" Type a "y" to change the line. If you type an "s" or end-of-file signal, the command will terminate. Other characters will cause a search for the next line containing <string1>.

Examples

cchange/ALPHA/OMEGA/!*	Perform a Controlled Change on all occurrences of "ALPHA" through the rest of the file
cc;a;z;-20 3	Perform a Controlled Change on the third occurrence of "a" in the current and previous 19 lines

COPY

Syntax

copy [<destination-target> [<range-target>]]
co [<destination-target> [<range-target>]]

Description

Copies the current line through <range-target> and places the copied text after the <destination-target>. The default <destination-target> is 1, thereby placing a copy of the current line after the next line. The default <range-target> is 1, thereby copying only one line. After the command is executed, the current line pointer will be set to the new position of the last line copied. Some lines may be renumbered after a copy with no renumbering message issued.

Examples

co #18	Put a copy of the current line after line 18
copy #3 4	Copy four lines beginning with the current line and place them after line 3
co /check/ +/range/	After the next line which has the string "check", place a copy of each line starting with the current line through the line containing "range"

DELETE

Syntax

delete [<target>] d [<target>]

Description

Deletes the current line (and successive lines until the target is reached). After the command is executed, the current line will be the line following the last line deleted.

Examples

delete 5	Delete five lines (the current line and the next four lines)
d	Delete the current line
d /STRING/	Delete lines from the current line through the next line that contains the string "STRING"

EXPAND

Syntax

expand [<target>]
exp [<target>]

Description

The current tab character is expanded within all lines, beginning with the current line, continuing down to and including the line specified by <target>. Since tabs are normally expanded as lines are inserted into the file, this command is primarily of use when one has forgotten to define a tab character or has inserted a tab character with an "append," "overlay," or "change" command.

Examples

expand 100	Expand 100 lines starting with the current line
exp	Expand the current line

INSERT

Syntax

insert
i

Description

The editor will enter the input mode, prompting with line numbers (unless line numbers have been disabled, with the "numbers" command) and insert the lines below the current line. The editor remains in "insert" until you begin a line with the "lino" character or the end-of-file signal in column one. The editor treats any characters following the "lino" character as an editor command. (If you type the "line delete character," the editor does not re-issue the prompt.

If possible, the editor will number the inserted lines with an increment small enough to insert at least 10 lines between the current line and the next line. The editor will renumber lines following the inserted text if the inserted text line numbers overlap numbers already in the file. (The current line pointer is left at the last line inserted.)

You may insert lines at the top of the edit buffer by specifying a line number of zero.

This command may not be followed by another command on the same line.

Examples

insert Accept line input after the current line

Oi Insert at the top of the edit buffer.

INSERT

Syntax

insert <text>
i <text>

Description

Inserts <text> as a separate line below the current line of the file. Use a space as a separator following the command name. The line inserted becomes the current line. The editor may renumber text lines following the inserted text if the inserted line number overlaps line numbers already in the file.

This command may not be followed by another command on the same line.

Examples

I This below the current line of the file

insert everything after the first blank

MERGE

Syntax

merge

Description

Merges the current line and the line immediately following it into a single line. The merged line becomes the current line.

Examples

merge Merge the current line and the next line into a single line.

MOVE

Syntax

```
move [<destination-target> [<range-target>]]
mo [<destination-target> [<range-target>]]
```

Description

Moves the current line through <range-target> so that they follow the line specified by <destination-target>. The defaults for <destination-target> and <range-target> are both 1, so "move" without arguments interchanges the current line and the next line. After the command is executed, the current line pointer will be set to the new position of the last line moved. Some lines may be renumbered with no renumbering message issued.

Examples

```
move 3          Move the current line down three lines

mo #1 /TARGET STRING/     Move the current line and all lines
                          down thru the line containing
                          "TARGET STRING" after line 1

mo -/Program/ 5      Move five lines (including the current
                     line) up within the file so that they
                     follow a line containing the character
                     string "Program"

mo #10 -5       Move the current line and the four previous
                lines below line number 10
```

OVERLAY

Syntax

```
overlay[<delimiter>]
o[<delimiter>]
```

Description

This command prints the current line, then accepts a line of input (the overlay line). When the line is printed, all ASCII HT characters will be displayed as spaces so that the terminal cursor will not be artificially advanced. The overlay line will be positioned directly beneath the line printed out. Each character of the overlay that is different from the <delimiter> character (which defaults to a blank) will replace the corresponding character in the current line. The overlaid line will be printed if verify is "on". If the end-of-file signal is typed in response to the prompt for the overlay line, the current line will not be changed.

Examples

```
overlay
     25.00=THIP IS THE CORRENT LUNE.
   Overlay    S         U
     25.00=THIS IS THE CURRENT LINE.
```

OVERLAY

Syntax

```
overlay<d><text>
o<d><text>
```

Description

This command is similar to the previous form of the "overlay" command with these differences: (1) The current line is not printed. (2) The remainder of the command line (after the delimiter character) is taken as the overlay text.

Examples

```
overlay--- AT-------------------- NUMBER.
   25.00=THAT IS THE CURRENT LINE NUMBER.
```

PRINT

Syntax

print [<target>] p [<target>]

Description

Prints all lines from the current line through the line specified by <target>. By default, only the current line will be printed.

Examples

p	Print the current line
print 5	Print 5 lines starting with the current line
p -10	Print the current line and the nine previous lines
print *string*	Print all lines down thru the next line containing "string"
p -/string/	Print all lines up through the next previous line containing "string"

REPLACE

Syntax

replace [<target>]
r [<target>]

Description

This command deletes from the current line through <target>, then places the editor in input mode, putting the new lines into the area vacated. It is not necessary to enter the same number of lines as were deleted. The line numbers of the lines inserted will probably not be the same as those deleted. The current line pointer will be positioned at the last line inserted. By default, only the current line will be deleted. This command may not be followed by another command on the same line.

Examples

r	Replace the current line
replace 10	Replace 10 lines starting with the current line
r /TARGET STRING/	Replace all lines from the current line through the line containing "TARGET STRING"

TEXT

Syntax

=<text>

Description

Replaces the current line with the text that follows the equal sign. The current line pointer is not moved.

Examples

```
=THIS IS REPLACEMENT TEXT.
```

NULL

Syntax

(null)

Description

The null command (i.e., just a carriage return) prints the current line.

DISK COMMANDS

FLUSH

Syntax

flush

Description

The information above the current line in the edit buffer is written to the file containing the updated data and then deleted from the edit buffer. Use this command to make room in the edit buffer for large insertions.

Examples

flush	Flush information above the current line to updated file.
200flush	Flush information above line 200 to the updated file.

NEW

Syntax

new

Description

The information above the current line in the edit buffer is written to the file containing the updated data and then deleted from the edit buffer. The available space in the edit buffer is then filled with data read from the file being edited. This command is used primarily to proceed to the next segment of the file when modifications to the current edit buffer have been completed. If a new file is being created, the "new" command is the same as the "flush" command.

Examples

new	Write the information above the current line to the updated file and read more data from the file being edited.
!new	Write the current edit buffer (except for the first line) to the updated file and read the next segment from the file being edited into the edit buffer.

READ

Syntax

read [<file name>]

Description

Places the contents of the specified file after the current line. The last line of the information read becomes the current line. If you omit the file name, the editor prompts you for it. If you type an end-of-file signal in response to the prompt, no data is read. The file name may contain path information if any is necessary to locate the file. The entire contents of the file must fit into the remaining unused space in the edit buffer. If the file being read will not fit into the edit buffer, the message "Not enough room" is issued and no data is read.

Examples

read /dde/data	Reads the information in the file "/dde/data" and places it after the current line.
100read moredata	Read the information in the file "moredata" and place it after line 100.

WRITE

Syntax

write [<target>]

Description

The editor prompts you for a file name, then writes the information from the current line through <target> to a file. If an end-of-file signal is typed in response to the prompt, no information is written. If the file being written already exists, it is destroyed and a new file created. If no <target> is specified, only the current line is written.

Examples

write /window/	Write the information from the current line through the line containing the string "window".
100write #200	Write lines 100 through 200, inclusive, to a scratch file.

EDITOR MESSAGES

A task is already running

The "x" command was used when there was already a task generated by a previous "x" command still running. The "wait" command must be used to wait for the previous task to complete before initiating another background task.

Attempting to merge onto last line of text

The "merge" command joins the specified line with the following line, and if the specified line is the last line of the file, there is no line following the specified line to join with it.

Bottom of file reached

An informative message issued when the last line of the file is deleted.

Cannot create configuration file

A configuration file could not be created in the directory specified in the "esave" command (current directory if no directory was mentioned). Usually this means that the directory specified could not be found or you don't have write permissions on that directory. Make sure the directory was specified with a trailing '/' character.

Cannot create new file

The editor was called with two file names as arguments, but the second file could not be created. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor. This message occurs only at the beginning of an editing session.

Cannot create new backup file

The editor detected an error attempting to create a backup file. This message is preceded by a message indicating which error was detected. The new backup file is not created and the editing session continues.

Cannot create task

An error was detected when trying to generate a task with the "u" or "x" command. This message is preceded by a message indicating which error was detected. The command is aborted and the editor requests a new command.

Cannot create temporary file

The editor detected an error when trying to create the temporary file that holds the updated information. This message is preceded by a message indicating which error was detected. This message occurs only at the beginning of an editing session.

Cannot delete old backup file

At the end of an editing session, the editor attempts to create a backup file containing the information as it was prior to the editing session. However, a file already exists with the backup file name, and that file could not be deleted. This message is preceded by a message indicating which error was detected. The new backup file is not created and the editing session continues.

Cannot open configuration file

The configuration file in the directory specified in an "eset" command could not be opened. This usually means that there was no configuration file in the specified directory, or that the specified directory could not be found, or that you do not have read permission for the configuration file. Remember that the directory name must be specified with a trailing '/' character.

Cannot open edit file

The file that is being edited exists, but could not be opened. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor. This message occurs only at the beginning of an editing session.

Cannot open new file

The editor was called with two file names as arguments, but could not open the second file to determine if it already exists. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor. This message occurs only at the beginning of an editing session.

Cannot read configuration file

The operating system reported a media error while the editor was trying to read from the editor configuration file.

Cannot read edit file

The operating system reported a media error while the editor was reading from the file whose data is being edited.

Cannot rename files

The editor detected an error trying to rename the files at the end of an editing session. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor. The user should then search for the temporary file used by the editor. This file will contain the updated information and should be copied to another file for safe keeping.

Cannot write configuration file

The operating system reported a media error while the editor was writing configuration data to the configuration file in the specified directory (current directory if the specification was omitted).

Delete existing backup file?

At the end of an editing session, the editor attempts to create a backup file containing the information as it was prior to the editing session. However, a file with the same name as the backup file would have already exists. This message is a request for permission to delete the existing file, replacing it with the new backup file. The prompt must be answered with a "y", for "yes", or an "n", for "no". If "y" or the end-of-file signal is typed, the file is deleted and the new backup file is created. If "n" is typed, the file will not be deleted and no new backup file created. If none of these are typed, the prompt is re-issued.

Delete existing copy of new file?

The editor was called with two file names as arguments. The second file already exists and must be deleted before the editing session can continue. This message is a request for permission to delete the file. The prompt must be answered with a "y", for "yes", or an "n", for "no". If "y" is typed, the file is deleted and the editing session continues. If "n" or the end-of-file signal is typed, the file will not be deleted and the editing session is terminated. If none of these are typed, the prompt is re-issued.

Edit file does not exist

The editor was called with two filenames, but the first file, which contains the data to be edited, could not be found. The editor will terminate immediately.

Empty text buffer

The text buffer is empty (contains no text) and the requested command could not be completed.

Error attempting to open file

The file specified in a "write" command could not be opened for writing. This usually means that the specified file could not be created because the path to the file was inaccessible, or the permissions on the directory in which the file was to reside exclude the you from creating a file there, or the file exists but the you do not have write permission for the file.

Error copying edit file

At the end of an editing session, any unread data on the file that is being edited is copied to the new file being written. An error was detected during this copy process. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor.

Error creating scratch file

The file specified in a "write" command could not be created. This message is preceded by a message indicating which error was detected. The "write" command is aborted and the editor requests a new command.

Error opening scratch file

The file specified in a "read" command could not be opened. This message is preceded by a message indicating which error was detected. The "read" command is aborted and the editor requests a new command.

Error reading data file

The editor detected an error when trying to read from the file being edited or from a scratch file with the "read" command. This message is preceded by a message indicating which error was detected. The current command is aborted and the editor requests a new command; no data read from the file is kept. If the file being read was the file being edited, you should use the "abort" command to abandon the editing session since the file being read is no longer positioned correctly.

Error waiting for task to complete

An error was detected when waiting for a task generated by the "u" or "x" command to complete. This message is preceded by a message indicating which error was detected. The command is aborted and the editor requests a new command.

Error writing new file

The editor detected an error when trying to write the contents of the edit buffer to the file that holds the updated information. This message is preceded by a message indicating which error was detected. This is a fatal error and will cause an immediate exit from the editor. All changes to information still in the edit buffer are lost.

File is a directory

An attempt was made to edit a directory, not a text file. This is a fatal error and causes an immediate exit from the editor. This message occurs only at the beginning of an editing session.

File name?

This is the prompt used when the editor requests a file name. Commands that may request a file name are "read" and "write". The editor will also request a file name in response to the "stop" and "log" commands if no file names were specified when the editor was called.

Input error

An error status was returned by the operating system in response to a request for input from the standard input device. This is normally the terminal keyboard and should not generate any such error. If the standard input has been redirected to a disk file, an error may be generated when reading the disk for input characters. In either case, this is a fatal error and causes an immediate exit from the editor. All changes to information still in the edit buffer are lost.

Line too long

The maximum size for a line being input to the editor is 255 characters. Lines in the file being edited may be of any length, but those entered from the standard input device are limited to 255 characters.

Name too long

The file name entered in response to a "File name:" prompt is too long. The maximum size of a file name, including the path specification, is 254 characters.

New file being created

This is an informative message indicating that there is no existing file of information to be edited and that a new file is being created.

New file is the same as the old file

The editor was called with two file names as arguments, but both names point to the same file. Either the file names are the same, or the two files have been linked with the "link" system call.

No child task exists

The "wait" command was used when no background task had been generated by the editor.

No lines deleted

An informative message indicating that the "delete" command was used but the target could not be located, and you answered "no" to the prompt asking if the delete was to proceed.

No such line

A line number or target could not be found.

Not enough room

The file being read with the "read" command could not fit in the available space in the edit buffer. None of the information read from the file is kept. You can use the "flush" command to try to make room for the file. If that fails, the file being read should be split into smaller files that may be read individually.

Not found

A target could not be found.

Output error

An error status was returned by the operating system in response to a request to send output to the standard output device. This is normally the terminal display and should not generate any such error. If the standard output has been redirected to a disk file, an error may be generated when writing the data to the disk file. In either case, this is a fatal error and causes an immediate exit from the editor. All changes to information still in the edit buffer are lost.

Positioning backwards is not allowed

The "position" command was called with a target that has a leading minus sign, indicating a backward search.

Relative positioning is not allowed

The "position" command was called with a target that is an unsigned integer, indicating a relative displacement forward in the file.

Some lines renumbered

An "insert," "replace," or "break" command caused some lines in the file to be renumbered. Note that the "move" and "copy" commands will cause renumbering without this message being issued.

Source overlaps destination

With the "copy" or "move" commands the target line was within the range of data being copied or moved.

Syntax error

A syntax error was detected in a command. Check the "Editor Commands" part of this section for correct editor command syntax.

Target not reached

Are you sure? The "delete" command was used but the target could not be located. If you want the delete to proceed to the end of the edit buffer, answer this prompt with a "y". Answering with an "n" or the end-of-file signal will cause the delete to be aborted.

Task ttt: Abnormal Termination

Interrupt code: i The background task "ttt" generated by the "x" command was interrupted before it could complete. The interrupt code returned by the task is indicated by "i". This message is returned only in response to the "wait" command.

Task ttt: Abnormal Termination

Termination response: xxx The background task "ttt" generated by the "x" command has completed abnormally. The termination response returned by the task is indicated by "xxx". This message is returned only in response to the "wait" command.

Task ttt initiated

Task number "ttt" has been started by the use of the "x" command.

Task ttt: Normal termination

The background task "ttt" generated by the "x" command has completed normally. This message is returned only in response to the "wait" command.

Too many file names specified

More than two file names were specified as arguments to the editor. This is an informative message only; the extra file names and any options specified after them are ignored.

Unable to open file

The file specified in a "read" command could not be found or could not be opened for reading because of its permissions.

Unexpected error, edit session aborted

An error response that the editor is incapable of handling was received from a system call . The editing session is terminated immediately.

Unknown option specified

An unrecognizable option was specified when the editor was called. This is an informative message only; the unrecognizable option is ignored.

Write ends with an error

The operating system reported a media error while the editor was writing data to the file specified in a "write" command.

...zones OK?

A target could not be found and the search zones were not set to their default values. This is an informative message asking you to check the zones because they may have been the reason that the target could not be found. This message does not require a response from you.

?

The editor is not able to interpret the given command. Either the command could not be recognized or the format of the command was undecipherable.

Section 10

TERMINAL EMULATION

OVERVIEW

When working on the 4404 you type on a keyboard and see messages displayed on a screen, just as with any terminal. When using "remote", the terminal emulator program, you can think of the entire 4404 as a terminal which is connected via an RS-232C line to a remote host computer. When you are using the 4404 as a stand-alone computer, you can think of the keyboard and display as a local terminal connected to the 4404 processor.

The 4404 appears to both the host and to it's internal software as an ANSI X3.64 compatible terminal with a few extensions that make it more compatible with other common ANSI X3.64 terminals.

The terminal emulator itself is a local terminal emulator which talks to the 4404 operating system's console driver. In conjunction with a local communication utility called "remote", the local terminal emulator console driver, and the driver for the communications port combine to create a remote terminal emulator connected to the RS-232 hardware and device driver. This makes the entire unit appear to an external host as a terminal.

This section contains a brief description of the appearance of the ANSI terminal emulator, a discussion of the interface between the emulator and the operating system, information on its default modes, and a description of how non-ASCII keys are handled. The section is concluded by a list and short description of all the implemented ANSI commands.

DESCRIPTION

The terminal emulator supports a display of 32 lines of 80 characters per line, using 8 by 15 pixel characters.

Compliance With ANSI and ISO Standards

The ANSI terminal emulator complies with the following ANSI (American National Standards Institute) and ISO (International Standards Organization) standards:

ANSI X3.4-1977,
American National Standard Code for Information Interchange.
(This defines the ASCII character set.)

ANSI X3.41-1974,
American National Standard Code Extension Techniques for Use With the 7-Bit Coded Character Set of American National Standard Code for Information Interchange. (This defines ways to extend the ASCII character set, including the exact way the SO and SI characters work to invoke G0 and G1 character sets.)

ISO 2022,
Code Extension Techniques for use with the ISO 7-bit Coded Character Set. (This is the international standard which corresponds to ANSI X3.41.)

ANSI X3.64-1979,
Additional Controls for Use With American Standard Code for Information Interchange. (This defines a variety of standard commands used for displaying text, editing the display of text, and for other functions.)

Compatibility with the DEC VT-100

The ANSI terminal emulator is NOT intended to emulate the VT-100. Some VT-100 DEC-private features which are of use to host editors have been included, but other DEC-private features have been omitted. Therefore, not all programs which run correctly with a VT-100 will run correctly with a 4404.

Compatibility with Tektronix Terminals

The ANSI terminal emulator is also NOT intended to emulate any of the Tektronix 4100 Series terminals. Many of the 4100 Series ANSI mode commands have been included, but some have been intentionally omitted.

INTERFACE TO THE OPERATING SYSTEM

The interface to the 4404 operating system is with the ttyget/ttyset system calls. These system calls are used to examine or modify the programmable modes of the emulator. This includes such things as autowrap on/off, screen normal/reverse, keypad application/numeric, cursor key application/numeric, LF/CR-LF, and tab locations.

The programmable modes of the emulator, mentioned above, all have default states which are specified in the discussion on ANSI commands. These defaults can be overridden by sending ANSI escape sequences to the terminal, or by using a ttyset system call (as in the "termset" utility).

The standard output of the non-ASCII keys on the keyboard (the function keys, the break-key, the keypad keys, and the joydisk) is an ANSI escape sequence (see the discussion on non-ASCII keys).

SUPPORTED ANSI COMMANDS

The following ANSI commands are supported on the 4404 terminal emulator:

NOTE

The ANSI <CSI> (control sequence identifier) is the two character sequence <Esc [>. In this discussion, it is represented as <CSI>.

<ACK> Acknowledge Character (char #6)

Syntax Form: (char #6)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<BEL> Bell Character

Syntax Form: (char #7)

Description: Sounds the terminal's bell.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

<BS> Backspace Character

Syntax Form: (char #8)

Description: BS moves the active position backward by one character position. If the cursor is already at column 1, then BS has no effect.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

<CAN> Character (#24)

Syntax Form: (Char #24)

Description: If this control character is received during an ANSI command sequence this control function will print a snoopy <CAN> character and resets the command parser to an initialized state.

<CBT> Cursor Backward Tab

Syntax Form: <CSI> [Pn] Z

Descriptive Form: <CSI> [desired number of preceding tab stops] Z

Description: Moves the cursor backwards to a preceding tab stop on the current line.

A parameter value of one moves the cursor to the preceding tab stop. A parameter value greater than one (N) moves the cursor to the Nth preceding tab stop on the current line. If there are less than N preceding tab stops, the cursor moves to column 1 of the current line.

If the parameter is zero or omitted, it defaults to 1.

<CHT> Cursor Horizontal Tab

Syntax Form: <CSI> [Pn] I

Descriptive Form: <CSI> [desired number of succeeding tab stops] I

Description: Moves the cursor forward to a succeeding tab stop on the current line.

A parameter value of one moves the cursor to the next tab stop. A value greater than one (N) moves the cursor to the Nth next tab stop on the current line. If there are less than N following tab stops, the cursor moves to the rightmost column of the current line.

If the parameter is zero or omitted, it defaults to 1.

<CPR> Cursor Position Report

Syntax Form: <CSI> <Pn> ; <Pn> R

Descriptive Form: <CSI> <row> ; <column> R

Description: The <CPR> message is sent from the terminal to the host in response to a <DSR: 6> "device status report" command.

If the origin mode is relative, the coordinates reported are "row, column" coordinates in the scrolling region. "Row 1, column 1" means the upper left corner of the region.

If the origin mode is absolute, the coordinates reported are "row, column" coordinates of the screen. "Row 1, column 1" means the upper left corner of the screen.

If the <CPR> is echoed back to the terminal, the terminal treats the echo as a no-op.

<CR> Carriage Return Character

Syntax Form: (char #13)

Description: Moves the cursor to the first column in the current line. If "carriage return/line feed" (CR/LF) mode is set, then a line feed action is also performed.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

<CRM> Control Representation Mode

Syntax Form: <CSI> 3 h or l

Descriptive Form: <CSI> 3 set or reset

Description: <CRM> is a parameter of the <RM> and <SR> commands.

This command is commonly referred to as a 'snoopy' mode.

Reset: Normal operation. <RM: CRM> resets this mode.

NOTE

The implementation of this command in the 4404 requires that <RM: CRM> not be embedded with other <RM> commands.

Set: "Snoopy" mode. CRM is set <SM: CRM>, commands are not interpreted, but rather the characters that make up the command are displayed.

Defaults: Reset

<CUB> Cursor Backward

Syntax Form: <CSI> [Pn] D

Descriptive Form: <CSI> [number of columns] D

Description: Moves the cursor backward by the specified number of columns. The cursor stops at column 1.

If the numeric parameter is 0 or is omitted, it defaults to 1.

<CUD> Cursor Down

Syntax Form: <CSI> [Pn] B

Descriptive Form: <CSI> [number of rows] B

Description: Moves the cursor downward by the specified number of rows.

Margins Set Inside Screen Boundaries
(i.e., Top Margin >1 or Bottom Margin <32)

If origin mode is absolute, the cursor moves with respect to the screen. If the cursor is on the last row of the screen or on the Bottom Margin, Cursor Down has no effect.

If origin mode is relative, the cursor moves with respect to the area bounded by Top and Bottom Margins. If the cursor is on the Bottom Margin, Cursor Down has no effect.

Margins Set To Screen Boundaries
(i.e., Top Margin =1 and Bottom Margin =32)

The cursor moves with respect to the screen. If the cursor is on the last row of the screen, Cursor Down has no effect.

If the <Pn> numeric parameter is zero or is omitted, it defaults to one.

<CUF> Cursor Forward

Syntax Form: <CSI> [Pn] C

Descriptive Form: <CSI> [number of columns] C

Description: Moves the cursor the specified number of columns to the right. The cursor stops at the rightmost column.

If the <Pn> numeric parameter is omitted, or is zero, it defaults to one.

<CUP> Cursor Position

Syntax Form: <CSI> [Pn] [; [Pn]] H

Descriptive Form: <CSI> [row number] [; [column number]] H

Description: Moves the cursor to a specified row and column. The cursor may stop at Top Margin, Bottom Margin and the top and bottom of the screen, depending on origin mode.

If a row or column coordinate is zero or is omitted, it defaults to one.

<CUU> Cursor Up

Syntax Form: <CSI> [Pn] A

Descriptive Form: <CSI> [number of rows] A

Description: This command is completely analogous to <CUD>, except that the cursor moves upward instead of downward.

<DA> Device Attributes

Syntax Form: <CSI> <Pn> c

Description: A device sends this command with a parameter of 0 to the terminal asking it to identify the type of VT100 terminal it is. The 4404 sends the command <CSI> ? 1 ; 0 c back to the device which says it is a VT100 with no options.

NOTE

The 4404 does support the following features of the VT-100 "Advanced Video Options:"

- o Bold
- o Underline
- o Reverse video

If the device echoes this command back to the terminal, it is treated as a no-op.

If the parameter is omitted, it defaults to 0.

<DC1> Character (Char #17)

Syntax Form: (Char #17)

Description: If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues. However if flagging is set in the communications system to DC1/DC3 flagging; a flagging action will occur within the communications system.

<DC2> Character (Char #18)

Syntax Form: (Char #18)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<DC3> Character (Char #19)

Syntax Form: (Char #19)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues. However if flagging is set in the communications system to DC1/DC3 flagging; a flagging action will occur within the communications system.

<DC4> Character (Char #20)

Syntax Form: (Char #20)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<DCH> Delete Character

Syntax Form: <CSI> [Pn] P

Descriptive Form: <CSI> [number of characters] P

Description: Deletes the character at the cursor and possibly following characters depending on the parameter value. Any characters to the right of the deleted characters are moved left by the same number of character positions; thus the gap is filled.

Only characters on the current line are affected by this command.

If the parameter is zero, or is omitted, it defaults to one.

<DEL> Character (Char #127)

Syntax Form: (Char #127)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<DL> Delete Line

Syntax Form: <CSI> [Pn] M

Descriptive Form: <CSI> [number of lines] M

Description: Deletes the current line and possibly succeeding lines, depending on the parameter.

All following lines are shifted in a block toward the line containing the cursor. The lines following the shifted portion are erased. The cursor does not change position.

If split-screen scrolling is in effect, this command only affects lines in the region that the cursor is currently in. (E.g., if the cursor is in the top fixed region, only the lines in the top fixed region are affected.)

If the parameter is zero, or is omitted, it defaults to one.

<DLE> Character (Char #16)

Syntax Form: (Char #16)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<DMI> Disable Manual Input

Syntax Form: Esc ` (Char #27 and Char #96)

Description: Locks the keyboard. This command is equivalent to ANSI <SM: KAM>

<DSR> Device Status Report

Syntax Form: <CSI> Ps n

Description: This is a command from the host or a report from the terminal. Table 10-1 shows the meaning of various parameters.

Table 10-1

PARAMETER MEANINGS

Parameter	Parameter Meaning
0	Report from 4404. Ready, no malfunctions detected.
3	Report from 4404. Malfunction - retry.
5	Command from host. Please report status (using a DSR control sequence).
6	Command from host. Please report cursor position (using a cursor position report). See <CPR> command.

When the 4404 receives a DSR with a parameter value of 5, it always sends back a DSR with a parameter value of 0 or 3. When the 4404 receives a DSR with a parameter of 6, it always sends back a CPR report. When the 4404 receives a DSR with a parameter value of 0 or 3 (which could be the echo of a report it has sent to the host), it executes the <DSR: 0> or <DSR: 3> command as a no-op.

<ECH> Erase Character

Syntax Form: <CSI> [Pn] X

Descriptive Form: <CSI> [number of characters] X

Description: Erases the character at the cursor, and possibly succeeding characters, according to the parameter. The cursor location remains unchanged.

The effect of the <ECH> command is not confined to the current line. For example, if the cursor is in column 41, and an <ECH: 45> command is issued, the character at the active position is erased along with the next 39 characters on the current line and the first 5 characters of the next line.

<ED> Erase in Display

Syntax Form: <CSI> [Ps] J

Descriptive Form: <CSI> [0 or 1 or 2] J

0 = from cursor to end of screen, inclusive
1 = from start of screen to cursor, inclusive
2 = entire screen.

Description: Regardless of whether margins are set, the command erases with respect to the screen. Therefore, text in the scrolling region and fixed regions can be erased with the same command.

The cursor does not change position.

If the parameter is omitted, it defaults to 0.

<EL> Erase in Line

Syntax Form: <CSI> [Ps] K

Descriptive Form: <CSI> [0 or 1 or 2] K

0 = from cursor to end of line, inclusive
1 = from start of line to cursor, inclusive
2 = entire line

Description: Erases part or all of the current line, according to the parameter. The cursor does not change position.

If parameter is omitted, it defaults to 0.

<EM> Character (Char #25)

Syntax Form: (Char #25)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<EMI> Enable Manual Input

Syntax Form: Esc b

Description: Unlocks the keyboard. This command is equivalent to ANSI <RM: KAM>

<ENQ> Character (Char #5)

Syntax Form: (Char #5)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<EOT> Character (Char #4)

Syntax Form: (Char #4)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence, this control action is a no-op and the ANSI command sequence processing continues.

<ESC> Character (Char #27)

Syntax Form: (Char #27)

Description: This control function is the introduction character of an escape sequence or control sequence for the ANSI command parser.

If this control character is received during an ANSI command sequence, the ANSI command sequence parser processing is reinitialized.

<ETB> Character (Char #23)

Syntax Form: (Char #23)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<ETX> Character (Char #3)

Syntax Form: (Char #3)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<FF> Form Feed Character

Syntax Form: (char #12)

Description: Page screen.

<FS> Character (Char #28)

Syntax Form: (Char #28)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<GS> Character (Char #29)

Syntax Form: (Char #29)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<HT) Horizontal Tab Character

Syntax Form: (char #9)

Description: Advances the cursor forward on the current line to the next horizontal tab stop. If there are no horizontal tab stops to the right of the active position, the cursor moves to the rightmost column.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

<HTS> Horizontal Tab Set

Syntax Form: ESC H

Description: Sets a tab stop at the current cursor location.

Defaults: Tab stops at columns 9, 17, 25, 33, 41, 49, 57, 65, and 73. Read from setup file on installation.

<HVP> Horizontal and Vertical Position

Syntax Form: <CSI> [Pn] [; [Pn]] f

Descriptive Form: <CSI> [row] [; [column]] f

Description: This command is identical to the <CUP>, Cursor Position command.

<ICH> Insert Character

Syntax Form: <CSI> [Pn] @

Descriptive Form: <CSI> [number of characters] @

Description: Inserts the specified number, (N), of erased character cells at the cursor position. The character currently at the cursor position and all other characters to the right of the cursor are shifted N columns to the right. Characters shifted off the end of the line are lost. The cursor position remains unchanged.

If the parameter is zero, or is omitted, it defaults to one.

<IL> Insert Line

Syntax Form: <CSI> [Pn] L

Descriptive Form: <CSI> [number of lines] L

Description: Inserts the specified number, (N), of blank lines in place of the active line.

The active line and all succeeding lines are shifted downwards. The last N lines of the scroll are lost. The cursor position does not change.

If split-screen scrolling is in effect, this command only affects lines in the region that the cursor is currently in. (E.g., if the cursor is in the scrollable (non-fixed) region, only the lines in the scrollable region are affected.)

If the parameter is zero or is omitted, it defaults to one.

<IND> Index

Syntax Form: ESC D

Description: Moves the active position down one line without affecting the character position on the line.

If the cursor is at the bottom margin, but is not at the bottom of the scroll, a scroll up function is performed. If the cursor is at the bottom margin and is also at the bottom of the scroll, a blank line is added to the bottom of the scroll and a scroll up is performed.

The cursor can index into the scrolling region from the top fixed region, but cannot index into bottom fixed region. An index on the last line of the bottom fixed region has no effect.

<IRM> Insertion/Replacement Mode

Syntax Form: <CSI> 4 h or l

Descriptive Form: <CSI> 4 set or reset

Description: < IRM> is a parameter for the <RM> and <SM> commands.

Reset: Normal operation. When a character is entered, it replaces any character already at the active position.

Set: Insert mode. As each character is entered, the text at the cursor position and to its right is moved one character cell to the right and the cursor advances to the next character cell. Any text which is shifted off the end of the line is lost.

Defaults: Reset

<KAM> Keyboard Action Mode

Syntax Form: <CSI> 2 h or l

Descriptive Form: <CSI> 2 set or reset

Description: A parameter for the <RM> and <SM> commands.

Reset: Resetting KAM enables the keyboard and is equivalent to issuing <EMI>.

Set: Setting KAM disables the keyboard and is equivalent to issuing <DMI>.

Defaults: Reset

<LF> Line Feed Character

Syntax Form: (char #10)

Description: If LNM mode is reset, then LF has exactly the same effect as the IND command; it advances the cursor to the same position on the following line of text. See the <IND> command description for details.

If LNM mode is set, then LF has the same effect as CR IND; it advances the active position to the first character position on the following line.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

<LNM> Line-Feed/New-Line Mode

Syntax Form: <CSI> 20 h or l

Descriptive Form: <CSI> 20 set or reset

Description: A parameter for the <RM> and <SM> commands.

Reset: (LF) is equivalent to <IND>; goes down one line without changing character position within the line.

Set: (LF) is equivalent to <NEL> (which is equivalent to (CR)<IND>). Advances the cursor to the first character position of the next line of text.

Defaults: Reset

<NAK> Character (Char #21)

Syntax Form: (Char #21)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<NEL> Next Line

Syntax Form: ESC E

Description: Moves the cursor to the start of the next line. Has the same effect as (CR)<IND> (or as (LF) when LNM is set).

<NUL> Character (Char #0)

Syntax Form: (Char #0)

Description: This control function is a no-op. If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<PU1> Private Use 1

Syntax Form: ESC Q

Description: This two-character sequence is used to introduce a private ANSI control sequence. It introduces all sequences which specify or request from 4404 reports on the state of the mouse buttons and the graphic cursor position.

<Report-Syntax-Mode>

Syntax Form: ESC # ! 0

Description: This command sends a 4100 series terminal <terminal-settings-report> to the host on the status of the syntax mode. The form will always be the following:

% ! <SP> <SP> 1 <CR>

NOTE

The <SP> is an ASCII space character. The <CR> (ASCII Carriage Return Character) is the default 4100 series EOM character.

<RI> Reverse Index

Syntax Form: ESC M

Description: Completely analogous to the IND (Index) command except that it moves the cursor one line upward.

<RIS> Reset to Initial State

Syntax Form: ESC c

Description: Resets specified terminal attributes to their initial default states.

This command affects terminal attributes in the following way:

- Erases screen and moves cursor to home position.
- Resets Insert/Replace mode to Replace.
- Clears edit margins.
- Turns off the character graphic rendition.
- Selects the default G0 and G1 character sets.
- Shifts in the G0 character set.
- Resets Auto-Repeat (TEKARM) mode := true.
- Resets Auto-Wrap (TEKAWM) mode := true.
- Resets Screen mode (TEMSCNM) to normal.
- Sets Origin mode to relative.

<RM> Reset Mode

Syntax Form: <CSI> [Ps] l

Description: Causes one or more modes to be reset, as specified by each selective parameter in the <Ps> parameter list. Each mode to be reset is specified by a separate parameter in the list. A mode is reset until set again by a <SM>, Set Mode, control sequence.

If the first character in the parameter list is "?", then all subsequent parameters, that consist of numeric digits only, are interpreted as if they began with a "?" character before those numeric digits. If the first parameter consists ONLY of "?", then its only use is to provide an implicit "?" at the start of each subsequent numeric-digits-only parameter in the parameter list.

For example:

The control sequence: <CSI> ? 5 ; 8 1
Is interpreted as if it were: <CSI> ? 5 ; ? 8 1

The control sequence: <CSI> ? ; 5 ; 8 1
Is interpreted as if it were: <CSI> ? 5 ; ? 8 1

Table 10-2 summarizes the meaning of the valid parameters.

Table 10-2

VALID RESET MODE PARAMETERS

Parameter	**Mode**
2	KAM Keyboard-Action-Mode.
3	CRM Control-Representation-Mode.
4	IRM Insertion-Replacement mode.
1 2	SRM Send/Receive mode.
2 0	LNM Line-Feed/New-Line mode.
? 1	TEKCKM TEK private Cursor Key mode.
? 5	TEKSCNM TEK private Screen mode (normal).
? 6	TEKOM TEK private Origin Mode (viewport)
? 7	TEKAWM TEK private Auto-Wrap mode.
? 8	TEKARM TEK private Auto-Repeat mode.

Any parameters other than those specified here are recognized and ignored.

<RS> Character (Char #30)

Syntax Form: (Char #30)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<SCS> Select Character Set

Syntax: <SCS> = <designate-G0-set> or <designate-G1-set>.
<its designate-G0-set> = (ESC) (() <set-selector>.
<designate-G1-set> = (ESC) ()) <set-selector>.
<set-selector> = (A)or(B)or(0)or(1)or(2)or(3).

Description: Designates a particular character set as the G0 set or the G1 set.

Table 10-3 summarizes the escape sequences necessary to designate particular character sets.

Table 10-3

CHARACTER SET SELECTION

Escape Sequence to Designate a G0 Set	**Escape Sequence to Designate a G1 Set**	**Character Set Being Designated As G0 Or G1**
ESC (A	ESC) A	(no-op)
ESC (B	ESC) B	U.S. (ASCII)
ESC (0	ESC) 0	Rulings
ESC (1	ESC) 1	(no-op)
ESC (2	ESC) 2	(no-op)
ESC (3	ESC) 3	Supplementary

Defaults: On installation, the terminal emulator automatically designates the U.S. (ASCII) character set as its G0 and G1 character set.

<Select-Code>

Syntax Form: ESC % ! <code-selector>

Description: This control function is a no-op.

<SGR> Select Graphic Rendition

Syntax Form: <CSI> [Ps-list] m

Parameters: The Ps-list consists of zero or more "Ps" selective parameters, separated by semicolons. Each parameter in the list specifies a graphic rendition for subsequent characters.

Description: Invokes the graphic rendition specified by the parameters in the Ps-list parameter string. All following characters in the data stream are displayed according to the parameter(s) until the next occurrence of an <SGR> command in the data stream.

In Tektronix terminals, each occurrence of the <SGR> control function causes only those graphic rendition aspects to be changed that are specified by that <SGR>. All other graphic rendition aspects remain unchanged. (In other words, the GRAPHIC RENDITION COMBINATION MODE of ISO 6429 is always set to CUMULATIVE in Tektronix terminals.)

Parameters are:

(0) Default rendition. In the 4404, "default rendition" is:

No underscore, normal boldness, standard (not reversed) image. That is, the effect of any preceding <SGR: 1>, <SGR: 4> or <SGR: 7> command is canceled.

(1) Bold or increased intensity: The 4404 represents this by simulating a bold font (it paints each character twice, shifted one pixel horizontally).

(4) Underscore.

(7) Negative (reverse) image: black characters on white background.

(2)(1) Not bold. Cancels the effect of <SGR: 1>.

(2)(4) Not underlined. Cancels the effect of <SGR: 4>.

(2)(7) Positive image. Cancels the effect of <SGR: 7>.

Defaults: An omitted parameter in the <Ps-list> defaults to zero. The state is that of <SGR: 0>.

<SI> Shift In Character

Syntax Form: (char #15)

Description: Invokes the current GO character set.

If this control character is received during an ANSI command sequence, the GO character is invoked and the ANSI command sequence processing continues.

Defaults: The GO set is invoked.

<SM> Set Mode

Syntax Form: <CSI> [Ps] h

Description: Causes one or more modes to be set, as specified by each selective parameter in the <Ps> parameter list. Each mode to be set is specified by a separate parameter. A mode is set until reset by a <RM> (Reset Mode) control sequence.

If the first character in the parameter list is "?", then all subsequent parameters, that consist of numeric digits only, are interpreted as if they began with a "?" character before those numeric digits. If the first parameter consists ONLY of "?", then its only use is to provide an implicit "?" at the start of each subsequent numeric-digits-only parameter in the parameter list.

For example:

The control sequence: <CSI> ? 5 ; 8 h
Is interpreted as if it were: <CSI> ? 5 ; ? 8 h

The control sequence: <CSI> ? ; 5 ; 8 h
Is interpreted as if it were: <CSI> ? 5 ; ? 8

Table 10-4 summarizes the meanings of the valid parameters to the "set mode" command.

Table 10-4

SET MODE PARAMETERS

Parameter	Mode
2	KAM Keyboard-Action-Mode.
3	CRM Control-Representation-Mode.
4	IRM Insertion-Replacement Mode.
1 2	SRM Send/Receive Mode.
2 0	LNM Line-Feed/New-Line Mode.
? 1	TEKCKM TEK private Cursor Key Mode.
? 5	TEKSCNM TEK private Screen Mode (Normal).
? 6	TEKOM TEK private Origin Mode (viewport)
? 7	TEKAWM TEK private auto-wrap mode.
? 8	TEKARM TEK private auto-repeat mode.

If no parameter is supplied, a parameter of zero is assumed. Any parameters other than those specified here (including zero) are recognized and ignored.

<SO> Shift Out Character

Syntax Form: SO = (char #14)

Description: Invokes the G1 character set. If this control character is received during an ANSI command sequence, the G1 character set is invoked and the ANSI command sequence processing continues.

Defaults: The G0 set is invoked (default is SO state).

<SOH> Character (Char #1)

Syntax Form: (Char #1)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<SP> "Space" Character

Syntax Form: (char #32)

Description: SP functions as an ordinary graphic character. Spaces replace any characters already in the locations where the spaces are typed.

<SRM> Send/Receive Mode

Syntax Form: <CSI> 1 2 h or l

Descriptive Form: <CSI> 1 2 set or reset

Description: < SRM>, Send/Receive Mode, is not a command in its own right. Rather, it is a parameter for the <SM>, Set Mode, and <RM>, Reset Mode, commands.

Resetting SRM mode turns the terminal's local echo on. (In the standards documents, this is called "monitor send/receive mode.")

Setting SRM mode turns the local echo off. (In the standards documents, this is "simultaneous send/receive mode.")

Defaults: Reset

<STX> Character (Char #2)

Syntax Form: (Char #2)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<SUB> Character (Char #26)

Syntax Form: (Char $26)

Description: If this control character is received during an ANSI command sequence this control function will print a (SUB) character and reset the command parser to an initialized state.

<SYN> Character (Char #22)

Syntax Form: (Char #22)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<TBC> Tabulation Clear

Syntax Form: <CSI> [Ps] g

Descriptive Form: <CSI> [0 or 2 or 3] g

Description: Clears one or more tab stops, according to the specified parameters.

The valid parameters are:

(0) Clear the horizontal tab stop at the active position.

(2) Clear all tab stops in the active line. (In the 4404 "tab stop mode" is always reset, so <TBC: 2> has the same effect as <TBC: 3>.)

(3) Clear all horizontal tab stops.

If no parameter is supplied, a parameter of zero is assumed. If the supplied parameter is not 0, 2 or 3, then command is ignored.

<TEKARM> Auto-Repeat Mode

Syntax Form: <CSI> ? 8 h or l

Descriptive Form: <CSI> ? 8 set or reset

Description: A TEK private parameter for the <SM> and <RM> commands. If set, all keyboard keys repeat when held depressed. If reset, none of the keys repeat when held depressed.

Defaults: Reset

<TEKAWM> Auto-Wrap Mode

Syntax Form: <CSI> ? 7 h or l

Descriptive Form: <CSI> ? 7 set or reset

Description: A TEK-private parameter for the <SM> and <RM> commands. When set, the wrap-around feature is enabled. When reset, it is disabled.

This mode determines what happens to the cursor after a character is displayed in the rightmost column. Since a character is always displayed at the current cursor location, this mode determines whether text is overprinted in the rightmost column or whether it wraps around to the next lines.

If Auto-Wrap mode is set, an index function is performed and the cursor moves to column 1 of the next line. If Auto-Wrap mode is reset, the cursor remains in the rightmost column.

Defaults: Reset

<TEKCKM> Cursor Key Mode

Syntax Form: <CSI> ? 1 h or l

Descriptive Form: <CSI> ? 1 set or reset

Description: A TEK private parameter for the <SM> and <RM> commands. Provides compatibility with programs designed for the DEC VT-100 terminal. This mode is only effective when TEKKPAM is set.

The joydisk keys assume the alternate meanings shown in Table 10-5.

Table 10-5

ALTERNATE JOYDISK MEANINGS

Joydisk Key	TEKCKM Reset	TEKCKM Set
Up	<CSI> A	ESC O A
Down	<CSI> B	ESC O B
Right	<CSI> C	ESC O C
Left	<CSI> D	ESC O D

Defaults: Reset

<TEKGCREP> Graphic Cursor Position Report

Syntax Form: ESC P S [Pn1 ; Pn2] ESC \ e

Descriptive Form: DCS S [optional position report] ST

Description: This is a report string sent to the host in response to a <TEKRGCR> graphic cursor position request. The form which the optional position report takes depends on the report types specified by a <TEKSGCRT> report type selection, or if report types have not been specified, by the default types defined there.

If cell coordinate reports have been specified, then Pn1 and Pn2 contain row and column values, respectively. If pixel coordinate reports have been specified, then Pn1 and Pn2 contain x and y screen coordinate values, respectively.

If graphics cursor position reports have been disabled by specifying that none be returned, then no parameters are returned for Pn1 and Pn2.

<TEKID> Identify Terminal

Syntax Form: ESC Z

This command, when sent from the host requests the terminal to identify itself with a Device Attributes sequence. It has the same effect as a <Device Attributes> command with no parameter or parameter of 0.

<TEKKPAM> Keypad Application Mode

Syntax Form: ESC =

Description: See <TEKKPNM>, Keypad Numeric Mode

<TEKKPNM> Keypad Numeric Mode

Syntax Form: ESC >

Description: The <TEKKPAM> and <TEKKPNM> commands set and reset the terminal's "Keypad Application Mode," respectively. These commands are provided for compatibility with applications programs designed for the DEC VT100 terminal.

Reset State (Keypad Numeric Mode)

In the "reset" state (Keypad Numeric Mode), the keypad keys and function keys F9 to F12 assume the values shown in the "reset state" part of the following table. For the keypad keys, these are the values labeled on the keys, except that the ENTER key sends a <CR> character.

Set State (Keypad Application Mode)

In the "set" state (Keypad Application Mode), the keypad keys and function keys F9 to F12 assume the alternate meanings shown in the "set state" part of Table 10-6.

Table 10-6

KEYPAD APPLICATION MODE KEY MEANINGS

Meaning in "Reset" State		Meaning in "Set" State
Key	Keypad Numeric Mode	Keypad Application Mode
0	0	ESC O p
1	1	ESC O q
2	2	ESC O r
3	3	ESC O s
4	4	ESC O t
5	5	ESC O u
6	6	ESC O v
7	7	ESC O w
8	8	ESC O x
9	9	ESC O y
-	-	ESC O m
,	,	ESC O l
.	.	ESC O n
ENTER	CR	ESC O M
F9	ESC O P	ESC O P
F10	ESC O Q	ESC O Q
F11	ESC O R	ESC O R
F12	ESC O S	ESC O S

Defaults: Reset (Keypad Numeric Mode)

<TEKMBREP> Mouse Button and Graphic Cursor Position Reporting

Syntax Form: DCS (Esc P)
Meta-State-Code
Mouse-Button-Number
Stroke-Info (up-down)
Optional-Position-Report
ST (Esc \)

Description: There are three buttons on the mouse and there are different codes output for each button on it's down-stroke and up-stroke.

Table 10-7 summarizes the Mouse Button Reports.

Table 10-7

MOUSE BUTTON REPORTS

Button	Left	Middle	Right
DOWN	DCS A 1 D x ST	DCS A 2 D x ST	DCS A 3 D x ST
UP	DCS A 1 U x ST	DCS A 2 U x ST	DCS A 3 U x ST
Shifted-DOWN	DCS B 1 D x ST	DCS B 2 D x ST	DCS B 3 D x ST
Shifted-UP	DCS B 1 U x ST	DCS B 2 U x ST	DCS B 3 U x ST
Control-DOWN	DCS C 1 D x ST	DCS C 2 D x ST	DCS C 3 D x ST
Control-UP	DCS C 1 U x ST	DCS C 2 U x ST	DCS C 3 U x ST
Cntrl-Shifted-DOWN	DCS D 1 D x ST	DCS D 2 D x ST	DCS D 3 D x ST
Cntrl-Shifted-UP	DCS D 1 U x ST	DCS D 2 U x ST	DCS D 3 U x ST

The "x" information is the optional report of the current graphic cursor position, i.e. "Pn1 ; Pn2" of the Graphic Cursor Report. See <TEKRGCR>.

<TEKOM> Origin Mode

Syntax Form: <CSI> ? 6 l or h

Parameters: l - Reset (Absolute Mode)
h - Set (Relative Mode)

Description: Margins Set To Screen Boundaries
(that is, Top Margin = 1, and Bottom Margin = 32)

Specifies Row 1, Column 1 of the screen as the origin. Moves the cursor to the origin.

Margins Set Inside Screen Boundaries
(i.e., Top Margin >1 or Bottom Margin < 32)

If origin mode absolute is requested, specifies Row 1, Column 1 of the screen as the origin. If origin mode relative is requested, specifies the row corresponding to the Top Margin, Column 1 as the origin. In both cases, it moves the cursor to the origin.

Defaults: Reset

<TEKRC> Restore Cursor

Syntax Form: ESC 8

Description: Restores the previously saved cursor position, graphic rendition, character set and origin mode.

If no preceding <Save Cursor> command has been executed, then the power-up graphic rendition, character set, and origin mode are restored and the cursor is homed.

<TEKREQTPARM> Request Terminal Parameters

Syntax Form: <CSI> [Pn] x

Description: Request from the host for the terminal to send a <Report Terminal Parameters> sequence. This command is treated as a no-op in the 4404.

<TEKRGCR> Request Graphic Cursor Position Report

Syntax Form: ESC Q K

Description: This command requests the terminal to send a report to the host as to the position of the graphics cursor. This report is a <TEKGCREP> report.

<TEKSC> Save Cursor

Syntax Form: ESC 7

Description: Saves the cursor position, graphic rendition, character set and origin mode.

<TEKSCNM> Screen Mode

Syntax Form: <CSI> ? 5 l or h

Parameters: l - Reset (Normal Mode -- white on black)
h - Set (Reverse Mode -- black on white)

Description: This is a parameter for the <Set Mode> and <Reset Mode> commands.

The reset state causes the screen to be black with white characters. The set state causes the screen to be white with black characters.

There is no effect if the terminal is already in the requested mode.

Defaults: Reset

<TEKSGCRT> Select Graphic Cursor Report Type

Syntax Form: ESC Q [Pn1] [;[Pn2]] J

Descriptive Form: ESC Q [Report When] [;[Report Type]] J

Parameter	Parameter Meaning
Pn1 = 0	None. Do not report mouse button action.
Pn1 = 1	Down. Report to host when mouse button is depressed.
Pn1 = 2	Up. Report to host when mouse button is released.
Pn1 = 3	Both. Report to host when a mouse button is either depressed or released.
Pn2 = 0	None. Do not report graphic cursor position.
Pn2 = 1	Char. Report graphics cursor position in character cell coordinate terms (Row, Column).
Pn2 = 2	Pixel. Report graphics cursor position in pixel (screen) coordinate terms (X,Y).

Defaults:

Pn1 = 0: No mouse button report.
Pn2 = 1: Report graphic cursor position in character cell coordinates

<TEKSTBM> Set Top and Bottom Margins

Syntax Form: <CSI> [Pn] [; [Pn]] r

Descriptive Form: <CSI> [top margin] [; [bottom margin]] r

Description: A TEK private command to set top and margins for a split viewport scrolling region.

The parameter value for the top margin specifies which row of the screen becomes the top line of the scrolling region. Similarly, the value for the bottom margin specifies the row of the buffer for the bottom line of the scrolling region.

The rows preceding the top margin and the rows following the bottom margin become fixed regions. No scrolling actions occur in the fixed regions.

If the first parameter is zero or is omitted, it defaults to one. If the second parameter is zero or is omitted, it defaults to 32.

Defaults: Margins set to 1 and 32

<US> Character (Char #31)

Syntax Form: (Char #31)

Description: This control function is a no-op.

If this control character is received during an ANSI command sequence this control action is a no-op and the ANSI command sequence processing continues.

<VT> Vertical Tab Character

Syntax Form: (char #11)

Description: VT has the same effect as (LF), linefeed.

If this control character is received during an ANSI command sequence this control action occurs and the ANSI command sequence processing continues.

ANSI Terminal Emulator Mouse Button and Position Reporting

Each of the three buttons on the mouse reports a different code on its downstroke and its upstroke. The mouse reports are ANSI standard DCS (Device Control String -- Esc-P) reports. The reports take the form:

DCS (Esc-P) -- Lead-in to all mouse button reports

Meta-State-Code -- A = unshifted, B = shifted, C = control, and D = control-shift

Mouse-Button-Number -- 1 = left, 2 = middle, 3 = right

Stroke-Info (up-down) -- D = down, U = up

Optional-Position-Report -- Pn1, Pn2 of the current position of the graphic cursor.

ST (ESC-\) -- Terminator for mouse button and position reports.

For example, the report (32;80 is the position report of Row 32, Column 80 in Char.Cell coordinates)) of the unshifted, middle button, in the down state would be:

DCS A 2 D 32;80 ST

<TEKSGCRT> Select_Graphic_Cursor_Report_Type (Tek-Private)

Syntax Form: ESC Q [Pn1] [; [Pn2]] J

Descriptive Form: ESC Q [Report When] [; [Report Type]] H

{ PU1 <Pn1> ; <Pn2> J }

Pn1 specifies: 0 - None, No mouse button reports.

1 - Down, Report to host when mouse button is depressed.

2 - Up, Report to host when mouse button is released.

3 - Both, Report to host when a button is depressed or released.

Powerup-Default - 0 - None

Pn2 specifies: 0 - None, No graphic cursor position report with mouse button reports.

1 - Char, The graphics cursor report is in character cell coordinate terms (Row,Column).

2 - Pixel, The graphics cursor report is in pixel (i.e.. screen) coordinate terms (X,Y).

Powerup-Default - 1 - Char

<TEKRGCR> Request Graphic Cursor Position Report (Tek Private)

Syntax Form: ESC Q K

Descriptive Form: ESC Q K { PU1 K }

This command sends a report to the host as to the position of the graphics cursor. The form of the report is as follows:

DCS (ESC P) S Pn1 ; Pn2 ST (ESC-\)

Pn1 contains: The Row value if cell coordinates have been selected or the X value if pixel coordinates are selected. no parameter will be returned if so specified in the Select Graphic Cursor Report Type command.

Pn2 contains: The Column value if cell coordinates have been selected or the Y value if pixel coordinates are selected. no parameter will be returned if so specified in the Select Graphic Cursor Report Type command.

KEYBOARD DETAILS

SHIFT, CTRL, AND CAPS LOCK KEYS

The two SHIFT keys have identical functions. They and the CTRL key are used to access alternate meanings for other keys.

Pressing CAPS LOCK turns on the led in the key and puts the keyboard in "caps lock mode." Pressing the key again turns the led off and removes the terminal from caps lock mode. While in caps lock mode, each of the alphabetic keys has its uppercase meaning, regardless of whether a SHIFT key is being held down. Caps lock mode affects only the alphabetic keys.

DEFAULT ANSI MODE MEANINGS OF KEYS

Alphanumeric Keys

Table 10-8 shows the ANSI mode meanings for the main part of the keyboard -- the "alphanumeric keys."

In this table, control characters are represented by the standard two-or three-letter abbreviations, given in ANSI X3.4 and ISO 646. Special symbols are represented by the four-character codes assigned to those symbols in ISO 6937. These meanings of these four-character codes are given in nearby notes.

Table 10-8

ANSI MEANINGS OF ALPHANUMERIC KEYS

Row 1 Keys (State)	**{ [**	**! 1**	**@ 2**	**# 3**	**$ 4**	**% 5**	**^ 6**
Unshifted	[	1	2	3	4	5	6
Shifted	{	!	@	#	$	%	^
Ctrl	ESC	1	2	3	4	5	6
Ctrl-Shifted	ESC	!	NUL	#	$	%	RS
R1 Keys	& 7	* 8	(9	) 0	SP09 SP10	+ =	}]	RUB
U	7	8	9	0	SP10	=	]	DEL
S	&	*	(	)	SP09	+	}	DEL
C	7	8	9	0	SP10	=	GS	DEL
C-S	&	*	(	)	US	+	GS	DEL

Row 2 Keys (State)	ESC		Q	W	E	R	T
Unshifted	ESC		q	w	e	r	t
Shifted	ESC		Q	W	E	R	T
Ctrl	ESC		DC1	ETB	ENQ	DC2	DC4
Ctrl-Shifted	ESC		DC1	ETB	ENQ	DC2	DC4
R2 Keys	Y	U	I	O	P	`	BS	LF
U	y	u	i	o	p		BS	LF
S	Y	U	I	O	P	`	BS	LF
C	EM	NAK	HT	SI	DLE	FS	BS	LF
C-S	EM	NAK	HT	SI	DLE	FS	BS	LF

Row 3 Keys (State)		TAB	A	S	D	F	G	H
Unshifted		HT	a	s	d	f	g	h
Shifted		HT	A	S	D	F	G	H
Ctrl		HT	SOH	DC3	EOT	ACK	BEL	BS
Ctrl-Shifted		HT	SOH	DC3	EOT	ACK	BEL	BS
R3 Keys	J	K	L	: ;	" '	RTN		
U	j	k	l	;	'	CR		
S	J	K	L	:	"	CR		
C	LF	VT	FF	;	'	CR		
C-S	LF	VT	FF	:	"	CR		

Row 4 Keys (State)		Z	X	C	V	B	N	M
Unshifted		z	x	c	v	b	n	m
Shifted		Z	X	C	V	B	N	M
Ctrl		SUB	CAN	ETX	SYN	STX	SO	CR
Ctrl-Shifted		SUB	CAN	ETX	SYN	STX	SO	CR
R 4 Keys	< ,	> .	? /					
U	,	.	/					
S	<	>	?					
C	,	.	/					
C-S	<	>	?					

Row 5 Keys -- Spacebar is "space" in all states.

Notes: SP09 = "low line" or underline
SP10 = hyphen or minus sign

Numeric Pad Keys

The numeric pad is located to the right of the main set of alphanumeric keys. The codes send by these keys are determined by the state of the Keypad Numeric/Applications mode setting (TEKKPNM/TEKKPAM). In Numeric mode, the meaning of the keys is that marked on the keytops; in Applications mode, the numeric pad keys are defined to be a control sequence. Table 10-9 shows the Applications mode (TEKKPAM) ANSI meanings of these keys.

Table 10-9

APPLICATIONS MODE (TEKKPAM) MEANINGS OF KEYPAD KEYS

Key Pad Name **State**	**0**	**1**	**2**	**3**	**4**
Unshifted	ESC O p	ESC O q	ESC O r	ESC O s	ESC O t
Shifted	ESC O p	ESC O q	ESC O r	ESC O s	ESC O t
Ctrl	ESC O p	ESC O q	ESC O r	ESC O s	ESC O t
Ctrl-Shifted	ESC O p	ESC O q	ESC O r	ESC O s	ESC O t

Key Pad Name State	5	6	7	8	9
Unshifted	ESC O u	ESC O v	ESC O w	ESC O x	ESC O y
Shifted	ESC O u	ESC O v	ESC O w	ESC O x	ESC O y
Ctrl	ESC O u	ESC O v	ESC O w	ESC O x	ESC O y
Ctrl-Shifted	ESC O u	ESC O v	ESC O w	ESC O x	ESC O y

Key Pad Name State	-	,	.	ENT	
Unshifted	ESC O m	ESC O l	ESC O n	ESC O M	
Shifted	ESC O m	ESC O l	ESC O n	ESC O M	
Ctrl	ESC O m	ESC O l	ESC O n	ESC O M	
Ctrl Shifted	ESC O m	ESC O l	ESC O n	ESC O M	

Joydisk Keys

The joydisk is located to the upper left of the main set of alphanumeric keys. The function of the joydisk in ANSI mode is to act in the place of cursor keys. The codes sent by the joydisk are affected by the Cursor Key mode in union with the Keypad Applications mode. The default codes are sent unless both TEKKPAM and TEKCKM are set. Table 10-10 shows the ANSI mode meanings of its keys.

Table 10-10

ANSI JOYDISK KEY MEANINGS

Joydisk Key Name	**Up**	**Down**	**Right**	**Left**
(Default mode)				
Unshifted	<CSI> A	<CSI> B	<CSI> C	<CSI> D
Shifted	<CSI> A	<CSI> B	<CSI> C	<CSI> D
Ctrl	<CSI> A	<CSI> B	<CSI> C	<CSI> D
Ctrl-Shifted	<CSI> A	<CSI> B	<CSI> C	<CSI> D
TEKKPAM and TEKCKM modes				
Unshifted	ESC O A	ESC O B	ESC O C	ESC O D
Shifted	ESC O A	ESC O B	ESC O C	ESC O D
Ctrl	ESC O A	ESC O B	ESC O C	ESC O D
Ctrl-Shifted	ESC O A	ESC O B	ESC O C	ESC O D

Function Keys

The function keys F1-F12 are grouped in three groups of four keys and are located in a row above both the alphanumeric keys and the numeric key pad. Table 10-11 shows the ANSI mode meanings of these keys.

Table 10-11

ANSI MEANINGS OF FUNCTION KEYS

Function Key Name vs. State	**F1**	**F2**	**F3**	**F4**
Unshifted	ESC O E	ESC O F	ESC O G	ESC O H
Shifted	ESC O E	ESC O F	ESC O G	ESC O H
Ctrl	ESC O E	ESC O F	ESC O G	ESC O H
Ctrl-Shifted	ESC O E	ESC O F	ESC O G	ESC O H
Function Key Name vs.State	F5	F6	F7	F8
Unshifted	ESC O I	ESC O J	ESC O K	ESC O L
Shifted	ESC O I	ESC O J	ESC O K	ESC O L
Ctrl	ESC O I	ESC O J	ESC O K	ESC O L
Ctrl-Shifted	ESC O I	ESC O J	ESC O K	ESC O L
Function Key Name vs. State	F9	F10	F11	F12
Unshifted	ESC O P	ESC O Q	ESC O R	ESC O S
Shifted	ESC O P	ESC O Q	ESC O R	ESC O S
Ctrl	ESC O P	ESC O Q	ESC O R	ESC O S
Ctrl-Shifted	ESC O P	ESC O Q	ESC O R	ESC O S

Special Function Keys

There are only two special function keys on the 4404 keyboard. One is the "up-arrow/left-arrow" key in the upper left corner of the main key area, while the other is the BREAK key in the lower right corner of the main key area. While most terminal emulators do not send a character sequence when the BREAK key is pressed, this emulator does -- under the assumption that the communication program will recognize the sequence and perform the appropriate break signal. Table 10-12 shows the default ANSI mode meaning of these keys.

Table 10-12

ANSI MEANINGS OF SPECIAL FUNCTION KEYS

Function Key Names vs. States		Break
Unshifted	ESC O T	ESC O @
Shifted	ESC O U	ESC O @
Ctrl	ESC O T	ESC O @
Ctrl- Shifted	ESC O U	ESC O @

Appendix A

ASCII CODE CHART

BITS B7 B6 B5 / B4 B3 B2 B1	Ø Ø Ø	Ø Ø 1	Ø 1 Ø	Ø 1 1	1 Ø Ø	1 Ø 1	1 1 Ø	1 1 1
	CONTROL		FIGURES		UPPERCASE		LOWERCASE	
Ø Ø Ø Ø	NU 0	DL 16	SP 32	0 48	@ 64	P 80	\ 96	p 112
Ø Ø Ø 1	SH 1	D1 17	! 33	1 49	A 65	Q 81	a 97	q 113
Ø Ø 1 Ø	SX 2	D2 18	" 34	2 50	B 66	R 82	b 98	r 114
Ø Ø 1 1	EX 3	D3 19	# 35	3 51	C 67	S 83	c 99	s 115
Ø 1 Ø Ø	ET 4	D4 20	$ 36	4 52	D 68	T 84	d 100	t 116
Ø 1 Ø 1	EQ 5	NK 21	% 37	5 53	E 69	U 85	e 101	u 117
Ø 1 1 Ø	AK 6	SY 22	& 38	6 54	F 70	V 86	f 102	v 118
Ø 1 1 1	BL 7	EB 23	' 39	7 55	G 71	W 87	g 103	w 119
1 Ø Ø Ø	BS 8	CN 24	(40	8 56	H 72	X 88	h 104	x 120
1 Ø Ø 1	HT 9	EM 25	) 41	9 57	I 73	Y 89	i 105	y 121
1 Ø 1 Ø	LF 10	SB 26	* 42	: 58	J 74	Z 90	j 106	z 122
1 Ø 1 1	VT 11	EC 27	+ 43	; 59	K 75	[91	k 107	{ 123
1 1 Ø Ø	FF 12	FS 28	, 44	< 60	L 76	\ 92	l 108	| 124
1 1 Ø 1	CR 13	GS 29	- 45	= 61	M 77	] 93	m 109	} 125
1 1 1 Ø	SO 14	RS 30	. 46	> 62	N 78	^ 94	n 110	~ 126
1 1 1 1	SI 15	US 31	/ 47	? 63	O 79	_ 95	o 111	DT 127

(4526)4893-18

INDEX

This material to be supplied later