NAME

intro, stdio – standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

The functions described in section 3S constitute a user-level I/O buffering scheme. The in-line macros *getc* and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher level routines *fgetc*, *getw*, *gets*, *fgets*, *scanf*, *fscanf*, *fread*, *fputc*, *putw*, *puts*, *fputs*, *printf*, *fprintf*, *fwrite* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **FILE**. *fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (−1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any module that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in sections labeled 3S of this manual are declared in that header file and need no further declaration. The constants and the following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *fileno*, and *clearerr*.

SEE ALSO

open(2V), close(2), lseek(2), pipe(2), read(2V), write(2V), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with *fopen*, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a *read*(2V) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use *read*(2V) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to *fflush* (see *fclose*(3S)) the standard output before going off and computing so that the output will appear.

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially *vfork*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
clearerr	ferror(3S)	stream status inquiries
ctermid	ctermid(3S)	generate filename for terminal
cuserid	cuserid(3S)	get character login name of user
fclose	fclose(3S)	close or flush a stream
fdopen	fopen(3S)	open a stream
feof	ferror(3S)	stream status inquiries
ferror	ferror(3S)	stream status inquiries
fflush	fclose(3S)	close or flush a stream
fgetc	getc(3S)	get character or integer from stream
fgets	gets(3S)	get a string from a stream
fileno	ferror(3S)	stream status inquiries
fopen	fopen(3S)	open a stream
fprintf	printf(3S)	formatted output conversion
fputc	putc(3S)	put character or word on a stream
fputs	puts(3S)	put a string on a stream
fread	fread(3S)	buffered binary input/output
freopen	fopen(3S)	open a stream
fscanf	scanf(3S)	formatted input conversion
fseek	fseek(3S)	reposition a stream
ftell	fseek(3S)	reposition a stream
fwrite	fread(3S)	buffered binary input/output
getc	getc(3S)	get character or integer from stream
getchar	getc(3S)	get character or integer from stream
gets	gets(3S)	get a string from a stream
getw	getc(3S)	get character or integer from stream
pclose	popen(3S)	initiate I/O to/from a process
popen	popen(3S)	initiate I/O to/from a process
printf	printf(3S)	formatted output conversion
putc	putc(3S)	put character or word on a stream
putchar	putc(3S)	put character or word on a stream
puts	puts(3S)	put a string on a stream
putw	putc(3S)	put character or word on a stream
rewind	fseek(3S)	reposition a stream
scanf	scanf(3S)	formatted input conversion
setbuf	setbuf(3S)	assign buffering to a stream
setbuffer	setbuf(3S)	assign buffering to a stream
setlinebuf	setbuf(3S)	assign buffering to a stream
sprintf	printf(3S)	formatted output conversion
sscanf	scanf(3S)	formatted input conversion
ungetc	ungetc(3S)	push character back into input stream
vfprintf	vprintf(3S)	print formatted varargs output
vprintf	vprintf(3S)	print formatted varargs output
vsprintf	vprintf(3S)	print formatted varargs output

NAME

ctermid – generate filename for terminal

SYNOPSIS

```
#include <stdio.h>
char *ctermid (s)
char *s;
```

DESCRIPTION

ctermid generates the pathname of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the **<stdio.h>** header file.

NOTES

The difference between *ctermid* and *ttyname*(3) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal. *ctermid* is useful largely for making code portable to non-UNIX systems where the current terminal is referred to by a name other than **/dev/tty**.

SEE ALSO

ttyname(3)

NAME

cuserid – get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

DESCRIPTION

cuserid generates a character-string representation of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (***e0**) will be placed at *s[0]*.

SEE ALSO

getlogin(3), getpwent(3)

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out, and the named *stream* to be closed. Buffers allocated by the standard input/output system are freed.

fclose is performed automatically for all open files upon calling *exit*(3).

fflush causes any buffered data for the named output *stream* to be written out. The named *stream* remains open.

SEE ALSO

close(2), exit(3), fopen(3S), setbuf(3S)

DIAGNOSTICS

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

ferror(stream)
FILE *stream;

feof(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

ferror returns non-zero when an error has occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

feof returns non-zero when EOF has previously been detected reading the named input *stream*, otherwise zero. Unless cleared by *clearerr*, the end-of-file indication lasts until the stream is closed.

clearerr resets the error indication and EOF indication to zero on the named *stream*.

fileno returns the integer file descriptor associated with the *stream*; see *open*(2V).

NOTE

All these functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3S), open(2V)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
char *type;
```

DESCRIPTION

fopen opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

filename points to a character string that contains the name of the file to be opened.

type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append: open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed, regardless of whether the open ultimately succeeds.

freopen is typically used to attach the preopened streams associated with **stdin, stdout,** and **stderr** to other files.

fdopen associates a stream with a file descriptor. File descriptors are obtained from calls like *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return streams. Streams are necessary input for many of the Section 3S library routines. The *type* of the stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

SEE ALSO

open(2V), fclose(3S), fseek(3S), fopen(3V)

DIAGNOSTICS

fopen and *freopen* return a NULL pointer on failure.

BUGS

In order to support the same number of open files as the system does, *fopen* must allocate additional memory for data structures using *calloc* after 20 files have been opened. This confuses some programs which use their own memory allocators.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

#include <stdio.h>

fread(ptr, size, nitems, stream)
FILE *stream;

fwrite(ptr, size, nitems, stream)
FILE *stream;

DESCRIPTION

fread reads, into a block pointed to by *ptr*, *nitems* of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. It returns the number of items actually read. *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream*.

If the standard output is line-buffered, *fread* flushes its output before reading from the standard input. *This is also true for the standard error.*

fwrite appends at most *nitems* of data from the block pointed to by *ptr* to the named output *stream*. It returns the number of items actually written. *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *fwrite* does not change the contents of the block pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

SEE ALSO

read(2V), write(2V), fopen(3S), getc(3S), putc(3S), gets(3S), puts(3S), printf(3S), scanf(3S), fread(3V)

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error.

NAME

fseek, ftell, rewind – reposition a stream

SYNOPSIS

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

DESCRIPTION

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

fseek and *rewind* undo any effects of *ungetc* (3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), ungetc(3S)

DIAGNOSTICS

fseek returns –1 for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen* (3S).

WARNING

Although on the UNIX system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

getc, getchar, fgetc, getw – get character or integer from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *Getc* and *getchar* are macros.

fgetc behaves like *getc*, but is a function rather than a macro. *fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next C **int** (word) from the named input *stream*. *getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *getw* assumes no special alignment in the file.

SEE ALSO

fopen(3S), putc(3S), gets(3S), ferror(3S), scanf(3S), fread(3S), ungetc(3S)

DIAGNOSTICS

These functions return the integer constant **EOF** at end-of-file or upon an error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return **EOF** until the condition is cleared with *clearerr*(3S). Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

BUGS

Because it is implemented as a macro, *getc* treats a *stream* argument with side effects incorrectly. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be readable using *getw* on a different processor.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

gets reads characters from the standard input stream, **stdin**, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character. *gets* returns its argument.

fgets reads characters from the *stream* into the array pointed to by *s*, until *n*−1 characters are read, a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character. *fgets* returns its first argument.

SEE ALSO

puts(3S), getc(3S), scanf(3S), fread(3S), ferror(3S)

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

popen, pclose – initiate I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;
```

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *popen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter, reading its standard input (which is also the standard input of the process doing the *popen*) and providing filtered input on the *stream*, and a type **w** command may be used as an output filter, reading a stream of output written to the *stream* process doing the *popen* and further filtering it and writing it to its standard output (which is also the standard input of the process doing the *popen*).

Popen always calls *sh*, never *csh*.

SEE ALSO

pipe(2), fopen(3S), fclose(3S), system(3), wait(2), sh(1)

DIAGNOSTICS

popen returns a NULL pointer if files or processes cannot be created, or the shell cannot be accessed.

pclose returns –1 if *stream* is not associated with a "*popen* ed" command.

BUGS

If the original and "*popen* ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
char *format;

char *sprintf(s, format [ , arg ] ... )
char *s, *format;

#include <varargs.h>
int _doprnt(format, args, stream)
char *format;
va_list *args;
FILE *stream;
```

DESCRIPTION

IX string "number conversion" string "number conversion — `printf`" *printf* places output on the standard output stream **stdout**. *fprintf* places output on the named output *stream*. *sprintf* places "output", followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. *printf* and *fprintf* return the number of characters transmitted, while *sprintf* returns a pointer to the string. *printf* and fprintf return an EOF if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** (ell) specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. A **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x or X** conversion, a non-zero result will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, unsigned decimal, or unsigned hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd" where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e,E The float or double *arg* is converted in the style "[–]d.ddde±ddd," where there is one digit before the decimal point and the number after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** or **E** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The **e**, **E**, **f**, **g**, and **G** formats print IEEE indeterminate values (infinity or not-a-number) as "Infinity" or "Nan" respectively.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

% Print a **%**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

NOTE

These routines call *_doprnt*, which is an implementation-dependent routine. Each uses the variable-length argument facilities of *varargs*(3). Although it is possible to use *_doprnt* to take a list of arguments and pass them on to a routine like *printf*, not all implementations have such a routine. We strongly recommend that you use the routines described in *vprintf (3S)* instead.

SEE ALSO

putc(3S), scanf(3S), ecvt(3), printf(3V)

BUGS

Very wide fields (>128 characters) fail.

The values "Infinity" and "Nan" cannot be read by *scanf*(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;
```

DESCRIPTION

putc writes the character *c* onto the named output *stream* (at the position where the file pointer, if defined, is pointing). It returns the character written.

putchar(*c*) is defined as *putc*(*c*, **stdout**). *putc* and *putchar* are macros.

fputc behaves like *putc*, but is a function rather than a macro. *fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

putw writes the C **int** (word) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. It returns the integer written. *putw* neither assumes nor causes special alignment in the file.

Output streams are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *setbuf*(3S), *setbuffer*(3S), or *setvbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fopen(3S), fclose(3S), getc(3S), puts(3S), printf(3S), fread(3S)

DIAGNOSTICS

On success, these functions each return the value they have written. On error, these functions return the constant **EOF** Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats a *stream* argument with side effects improperly. In particular, **putc(c, *f++);** doesn't work sensibly. *fputc* should be used instead.

Errors can occur long after the call to *putc*.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string pointed to by *s*, followed by a newline character, to the standard output stream **stdout**.

fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminal null character.

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

fopen(3S), putc(3S), printf(3S), ferror(3S), fread(3S)

NOTES

puts appends a newline while *fputs* does not.

NAME

scanf, fscanf, sscanf – formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf(format [, pointer ] ... )
char *format;

fscanf(stream, format [, pointer ] ... )
FILE *stream;
char *format;

sscanf(s, format [, pointer ] ... )
char *s, *format;
```

DESCRIPTION

scanf reads from the standard input stream **stdin**. *fscanf* reads from the named input *stream*. *sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, or new-lines) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion characters are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be a integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optional +, –, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

[indicates string data; the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be capitalized or preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E–1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0–9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3S) strtod(3), strtol(3), scanf(3V)

DIAGNOSTICS

These functions return **EOF** on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

scanf cannot read the strings which *printf*(3S) generates for IEEE indeterminate floating point values.

scanf provides no way to convert a number in any arbitrary base (decimal, hex or octal) based on the traditional **C** conventions (leading 0 or 0x).

NAME

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

setlinebuf(stream)
FILE *stream;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. *fflush* (see *fclose*(3S)) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from *malloc*(3) upon the first *getc* or *putc*(3S) on the file. If the standard stream **stdout** refers to a terminal it is line buffered. If the standard stream **stderr** refers to a terminal it is line buffered.

setbuf can be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered. A manifest constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

setbuffer, an alternate form of *setbuf*, can be used after a stream has been opened but before it is read or written. It causes the character array *buf* whose size is determined by the *size* argument to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered.

setvbuf can be used after a stream has been opened but before it is read or written. *type* determines how *stream* will be buffered. Legal values for *type* (defined in **<stdio.h>**) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered. If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used.

setlinebuf is used to change the buffering on a stream from block buffered or unbuffered to line buffered. Unlike *setbuf*, *setbuffer*, and *setvbuf*, it can be used at any time that the file descriptor is active.

A file can be changed from unbuffered or line buffered to block buffered by using *freopen* (see *fopen*(3S)). A file can be changed from block buffered or line buffered to unbuffered by using *freopen* followed by *setbuf* with a buffer argument of NULL.

SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3), fclose(3S), puts(3S), printf(3S), fread(3S), setbuf(3V)

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3), perror(3), tmpnam(3S)

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

tmpnam always generates a file name using the path-prefix defined as **P_tmpdir** in the **<stdio.h>** header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the **<stdio.h>** header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

tempnam uses *malloc* to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc* failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen* or *creat* are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink* to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3), mktemp(3), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>

ungetc(c, stream)
FILE *stream;
```

DESCRIPTION

ungetc pushes the character *c* back onto an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. In the case that *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

An *fseek*(3S) erases all memory of pushed back characters.

SEE ALSO

getc(3S), setbuf(3S), fseek(3S)

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

char *vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(3).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *    separately declared because of the definition of varargs.
 */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void)vfprintf(fmt, args);
        va_end(args);
        (void)abort( );
}
```

SEE ALSO

varargs(3)

NAME

intro – introduction to System V functions

SYNOPSIS

/usr/5bin/cc

DESCRIPTION

These functions are contained in the System V library, */usr/5lib/libc.a*. They are automatically linked when you compile a C program with the C compiler in **/usr/5bin/cc**.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
_tolower	ctype(3V)	character classification and conversion
_toupper	ctype(3V)	character classification and conversion
asctime	ctime(3V)	convert date and time to ASCII
assert	assert(3V)	verify program assertion
ctime	ctime(3V)	convert date and time to ASCII
curses	curses(3V)	CRT screen handling and optimization package
endpwent	getpwent(3V)	get password file entry
fdopen	fopen(3V)	open a stream
feof	ferror(3V)	stream status inquiry
ferror	ferror(3V)	stream status inquiry
fgetc	getc(3V)	get character or integer from stream
fgetpwent	getpwent(3V)	get password file entry
fileno	ferror(3V)	stream status inquiry
fopen	fopen(3V)	open a stream
fprintf	printf(3V)	formatted output conversion
fread	fread(3V)	buffered binary input/output
freopen	fopen(3V)	open a stream
fscanf	scanf(3V)	formatted input conversion
fwrite	fread(3V)	buffered binary input/output
getc	getc(3V)	get character or integer from stream
getchar	getc(3V)	get character or integer from stream
getpass	getpass(3V)	read a password
getpwent	getpwent(3V)	get password file entry
getpwnam	getpwent(3V)	get password file entry
getpwuid	getpwent(3V)	get password file entry
getw	getc(3V)	get character or integer from stream
gmtime	ctime(3V)	convert date and time to ASCII
isalnum	ctype(3V)	character classification and conversion
isalpha	ctype(3V)	character classification and conversion
isascii	ctype(3V)	character classification and conversion
iscntrl	ctype(3V)	character classification and conversion
isdigit	ctype(3V)	character classification and conversion
isgraph	ctype(3V)	character classification and conversion
islower	ctype(3V)	character classification and conversion
isprint	ctype(3V)	character classification and conversion
ispunct	ctype(3V)	character classification and conversion
isspace	ctype(3V)	character classification and conversion
isupper	ctype(3V)	character classification and conversion
isxdigit	ctype(3V)	character classification and conversion
localtime	ctime(3V)	convert date and time to ASCII
nice	nice(3V)	change priority of a process
printf	printf(3V)	formatted output conversion

rand	rand(3V)	simple random number generator
scanf	scanf(3V)	formatted input conversion
setbuf	setbuf(3V)	assign buffering to a stream
setbuffer	setbuf(3V)	assign buffering to a stream
setlinebuf	setbuf(3V)	assign buffering to a stream
setpwent	getpwent(3V)	get password file entry
setuid	setuid(3V)	set user ID
setvbuf	setbuf(3V)	assign buffering to a stream
signal	signal(3V)	simplified software signal facilities
sleep	sleep(3V)	suspend execution for interval
sprintf	printf(3V)	formatted output conversion
srand	rand(3V)	simple random number generator
sscanf	scanf(3V)	formatted input conversion
times	times(3V)	get process and child process times
toascii	ctype(3V)	character classification and conversion
tolower	ctype(3V)	character classification and conversion
toupper	ctype(3V)	character classification and conversion
ttyslot	ttyslot(3V)	find the slot in the utmp file of the current process
tzset	ctime(3V)	convert date and time to ASCII

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression; System V"

DESCRIPTION

assert is a macro that indicates *expression* is expected to be true at this point in the program. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the *cc*(1) option –DNDEBUG, or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cc(1), abort(3)

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

ctime converts to ASCII a long integer, pointed to by *clock*, that represents the time in seconds since Jan. 1, 1970, 00:00, Greenwich Mean Time. It returns a pointer to a 26-character string of the form:

Sun Sep 16 01:03:52 1973\n\0

Each field has a constant width. *localtime* and *gmtime* return pointers to structures containing the time broken down. *localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. *asctime* converts the broken-down time to ASCII and returns a pointer to a 26-character string.

Declarations of all the functions and externals, and the "tm" structure, are in the **<time.h>** header file. The structure declaration is:

```
struct tm {
    int tm_sec;         /* seconds (0 - 59) */
    int tm_min;         /* minutes (0 - 59) */
    int tm_hour;        /* hours (0 - 23) */
    int tm_mday;        /* day of month (1 - 31) */
    int tm_mon;         /* month of year (0 - 11) */
    int tm_year;        /* year – 1900 */
    int tm_wday;        /* day of week (Sunday = 0) */
    int tm_yday;        /* day of year (0 - 365) */
    int tm_isdst;
};
```

tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in PST, *timezone* is 8*60*60); the external variable *daylight* is non-zero if and only if Daylight Savings Time conversion should be applied. Its value indicates the type of conversion to apply; it is normally the value returned by *gettimeofday (2)* in the *tz_dsttime* field of the *timezone* structure. The program knows about various peculiarities in time conversion over the past 10-20 years.

The external variable *tzname* is an array of two pointers which contains the names of the current time zone. The first pointer points to a character string which is the name of the current time zone when Daylight Savings Time is not in effect; the second one, if Daylight Savings Time conversion should be applied, points to a character string which is the name of the current time zone when Daylight Savings Time is in effect.

If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the time zone and conversion rule type supplied by the system. The value of TZ must be a three-letter time zone name, followed by a signed number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for California would be **PST8PDT**. The effects of setting TZ are thus to change the values of the external variables *timezone*, *daylight*, and *tzname*. The function *tzset* sets these external variables from TZ or, if TZ is not present in the environment, the values supplied by the system. *tzset* is called by *asctime* and may also be called explicitly by the user.

SEE ALSO

gettimeofday(2), time(3C), getenv(3), environ(5V), ctime(3)

BUGS

The return values point to static data, whose contents are overwritten by each call.

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii, isgraph, toupper, tolower, toascii, _toupper, _tolower – character classification and conversion macros and functions

SYNOPSIS

#include <ctype.h>

isalpha(c)

...

CHARACTER CLASSIFICATION MACROS

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where isascii(*c*) is true and on the single non-ASCII value EOF (see *stdio*(3S)).

isalpha(c) *c* is a letter

isupper(c) *c* is an upper case letter

islower(c) *c* is a lower case letter

isdigit(c) *c* is a digit [0-9].

isxdigit(c) *c* is a hexadecimal digit [0-9], [A-F], or [a-f].

isalnum(c) *c* is an alphanumeric character, that is, *c* is a letter or a digit

isspace(c) *c* is a space, tab, carriage return, newline, vertical tab, or formfeed

ispunct(c) *c* is a punctuation character (neither control nor alphanumeric)

isprint(c) *c* is a printing character, code 040(8) (space) through 0176 (tilde)

iscntrl(c) *c* is a delete character (0177) or ordinary control character (less than 040).

isascii(c) *c* is an ASCII character, code less than 0200

isgraph(c) *c* is a visible graphic character, code 041 (exclamation mark) through 0176 (tilde).

CHARACTER CONVERSION MACROS AND FUNCTIONS

toupper and *tolower* are functions, rather than macros, and work correctly on all characters. The macros *_toupper* and *_tolower* are faster than the equivalent functions (*toupper* and *tolower*) but only work properly on a restricted range of characters.

These functions perform simple conversions on single characters.

toupper(c) converts *c* to its upper-case equivalent. If *c* is not a lower-case letter, it is returned unchanged.

tolower(c) converts *c* to its lower-case equivalent. If *c* is not an upper-case letter, it is returned unchanged.

toascii(c) masks *c* with the correct value so that *c* is guaranteed to be an ASCII character in the range 0 thru 0x7f.

These macros perform simple conversions on single characters.

_toupper(c) converts *c* to its upper-case equivalent. Note that this *only* works where *c* is known to be a lower-case character to start with (presumably checked via *islower*).

_tolower(c) converts *c* to its lower-case equivalent. Note that this *only* works where *c* is known to be a upper-case character to start with (presumably checked via *isupper*).

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

stdio(3S), ascii(7), ctype(3)

NAME

curses – CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>
/usr/5bin/cc [*flags*] *files* **–lcurses** [*libraries*]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling *initscr()* you should call "*nonl*(); *cbreak*(); *noecho*(); "

The full curses interface permits manipulation of data structures called *windows* which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, and others can be created with *newwin*. Windows are referred to by variables declared "WINDOW *", the type WINDOW is defined in **curses.h** to be a C structure. These data structures are manipulated with functions described below, among which the most basic are *move* and *addch*. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect **stdscr**.) Then *refresh* is called, telling the routines to make the user's CRT screen look like **stdscr**.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use -DMINICURSES as a **cc** option. This level is smaller and faster than full curses.

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/5lib/terminfo**, and TERM is set to "vt100", then normally the compiled file is found in **/usr/5lib/terminfo/v/vt100**. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to **/usr/mark/myterms**, curses will first check **/opusr/mark/myterms/v/vt100**, and if that fails, will then check **/usr/5lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/5lib/terminfo** is not available.

SEE ALSO

ioctl(2), getenv(3), tty(4), terminfo(5V)

FUNCTIONS

Routines listed here may be called when using the full curses. Those marked with an asterisk may be called when using Mini-Curses.

addch(ch)*	add a character to *stdscr* (like putchar) (wraps to next line at end of line)
addstr(str)*	calls addch with each character in *str*
attroff(attrs)*	turn off attributes named
attron(attrs)*	turn on attributes named
attrset(attrs)*	set current attributes to *attrs*
baudrate()*	current terminal speed
beep()*	sound beep on terminal
box(win, vert, hor)	draw a box around edges of *win* *vert* and *hor* are chars to use for vert. and hor. edges of box
clear()	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*
clrtobot()	clear to bottom of *stdscr*
clrtoeol()	clear to end of line on *stdscr*
cbreak()*	set cbreak mode

delay_output(ms)*	insert ms millisecond pause in output
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
doupdate()	update screen from all wnooutrefresh
echo()*	set echo mode
endwin()*	end window modes
erase()	erase *stdscr*
erasechar()	return user's erase character
fixterm()	restore tty to "in curses" state
flash()	flash screen or beep
flushinp()*	throw away any typeahead
getch()*	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode()	establish current tty modes
getyx(win, y, x)	get (y, x) co-ordinates
has_ic()	true if terminal can do insert character
has_il()	true if terminal can do insert line
idlok(win, bf)*	use terminal's insert/delete line if bf != 0
inch()	get char at current (y, x) co-ordinates
initscr()*	initialize screens
insch(c)	insert a char
insertln()	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar()	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*, otherwise cursor must be left at current position.
longname()	return verbose name of terminal
meta(win, flag)*	allow meta characters on input if flag != 0
move(y, x)*	move to (y, x) on *stdscr*
mvaddch(y, x, ch)	move(y, x) then addch(ch)
mvaddstr(y, x, str)	similar...
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	
mvwinsch(win, y, x, c)	
mvwprintw(win, y, x, fmt, args)	
mvwscanw(win, y, x, fmt, args)	

newpad(nlines, ncols)	create a new pad with given dimensions
newterm(type, fd)	set up new terminal of given type to output on fd
newwin(lines, cols, begin_y, begin_x)	create a new window
nl()*	set newline mapping
nocbreak()*	unset cbreak mode
nodelay(win, bf)	enable nodelay input mode through getch
noecho()*	unset echo mode
nonl()*	unset newline mapping
noraw()*	unset raw mode
overlay(win1, win2)	overlay win1 on win2
overwrite(win1, win2)	overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)	refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)	printf on *stdscr*
raw()*	set raw mode
refresh()*	make current screen look like *stdscr*
resetterm()*	set tty modes to "out of curses" state
resetty()*	reset tty flags to stored value
saveterm()*	save current modes as "in curses" state
savetty()*	store current tty flags
scanw(fmt, arg1, arg2, ...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win, flag)	allow terminal to scroll if flag != 0
set_term(new)	now talk to terminal new
setscrreg(t, b)	set user scrolling region to lines t through b
setterm(type)	establish terminal with given type
setupterm(term, filenum, errret)	
standend()*	clear standout mode attribute
standout()*	set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)	create a subwindow
touchwin(win)	"change" all of *win*
traceoff()	turn off debugging trace output
traceon()	turn on debugging trace output
typeahead(fd)	use file descriptor fd to check typeahead
unctrl(ch)*	printable version of *ch*
waddch(win, ch)	add char to *win*
waddstr(win, str)	add string to *win*
wattroff(win, attrs)	turn off *attrs* in *win*
wattron(win, attrs)	turn on *attrs* in *win*
wattrset(win, attrs)	set attrs in *win* to *attrs*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*

wdelch(win, c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win, str)	get a string through *win*
winch(win)	get char at current (y, x) in *win*
winsch(win, c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win, y, x)	set current (y, x) co-ordinates on *win*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in terminfo(4). The include files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parmeterized strings should be passed through *tparm* to instantiate them. All terminfo strings (including the output of tparm) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm()	restore tty modes for terminfo use (called by setupterm)
resetterm()	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to UNIX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be *setupterm(0, 1, 0)* which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar-like function to which the characters are passed, one at a time.
putp(str)	handy function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like function *putc*.
vidattr(attrs)	Like vidputs but outputs through putchar

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use *termcap*. Their parameters are the same as for *termcap*. They are emulated using the *terminfo* database. They may go away at a later date.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap

ATTRIBUTES

The following video attributes can be passed to the functions *attron*,*attroff*,*attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys might be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported, due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNING

The plotting library *plot*(3X) and the curses library *curses*(3V) both use the names erase() and move(). The curses versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3V) code, and/or #undef move() and erase() in the *plot*(3X) code.

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

ferror(stream)
FILE *stream;

feof(stream)
FILE *stream;

clrerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

ferror returns non-zero when an error has occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

feof returns non-zero when **EOF** has previously been detected reading the named input *stream*, otherwise zero. Unless cleared by *clearerr*, the end-of-file indication lasts until the stream is closed; however, operations which attempt to read from the stream will ignore the current state of the end-of-file indication and attempt to read from the file descriptor associated with the stream.

clearerr resets the error indication and **EOF** indication to zero on the named *stream*.

fileno returns the integer file descriptor associated with the *stream*; see *open*(2V).

NOTE

All these functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3S), open(2V)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
char *type;
```

DESCRIPTION

fopen opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

filename points to a character string that contains the name of the file to be opened.

type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append: open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed, regardless of whether the open ultimately succeeds.

freopen is typically used to attach the preopened streams associated with **stdin, stdout,** and **stderr** to other files.

fdopen associates a stream with a file descriptor. File descriptors are obtained from calls like *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return streams. Streams are necessary input for many of the Section 3S library routines. The *type* of the stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

open(2), fclose(3S), fseek(3S), fopen(3S)

DIAGNOSTICS

fopen and *freopen* return a NULL pointer on failure.

BUGS

In order to support the same number of open files as does the system, *fopen* must allocate additional memory for data structures using *calloc* after 20 files have been opened. This confuses some programs which use their own memory allocators.

NAME

fread, fwrite – buffered binary input/output

SYNOPSIS

#include <stdio.h>

fread(ptr, size, nitems, stream)
FILE *stream;

fwrite(ptr, size, nitems, stream)
FILE *stream;

DESCRIPTION

fread reads, into a block pointed to by *ptr*, *nitems* of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. It returns the number of items actually read. *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream*.

When input is read from *any* line-buffered stream, output to *all* line-buffered streams is flushed (including the standard error). Input read from a stream that is not line-buffered does not cause flushing of these streams.

fwrite appends at most *nitems* of data from the block pointed to by *ptr* to the named output *stream*. It returns the number of items actually written. *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *fwrite* does not change the contents of the block pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

SEE ALSO

read(2V), write(2V), fopen(3S), getc(3S), putc(3S), gets(3S), puts(3S), printf(3S), scanf(3S), fread(3S)

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error.

NAME

getc, getchar, fgetc, getw – get character or integer from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *Getc* and *getchar* are macros.

fgetc behaves like *getc*, but is a function rather than a macro. *fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

getw returns the next C **int** (word) from the named input *stream*. *getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *getw* assumes no special alignment in the file.

SEE ALSO

fopen(3S), putc(3S), gets(3S), ferror(3S), scanf(3S), fread(3S), ungetc(3S)

DIAGNOSTICS

These functions return the integer constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

BUGS

Because it is implemented as a macro, *getc* treats a *stream* argument with side effects incorrectly. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be readable using *getw* on a different processor.

NAME

getpass – read a password

SYNOPSIS

char *getpass(prompt)
char *prompt;

DESCRIPTION

getpass reads up to a newline or EOF from the file **/dev/tty**, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. An interrupt will terminate input and send an interrupt signal to the calling program before returning. If **/dev/tty** cannot be opened, a NULL pointer is returned; the standard input is not read.

FILES

/dev/tty

SEE ALSO

crypt(3), getpass(3)

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

int setpwent()

int endpwent()

struct passwd *fgetpwent(f)
FILE *f;
```

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file. Each line in the file contains a "passwd" structure, declared in the **<pwd.h>** header file:

```
struct    passwd { /* see getpwent(3) */
          char    *pw_name;
          char    *pw_passwd;
          int     pw_uid;
          int     pw_gid;
          char    *pw_age;
          char    *pw_comment;
          char    *pw_gecos;
          char    *pw_dir;
          char    *pw_shell;
};

          struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

This structure is declared in **<pwd.h>** so it is not necessary to redeclare it.

The field *pw_comment* is unused; the others have meanings described in *passwd*(5). When first called, *getpwent* returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of the password file **/etc/passwd**.

The field, *pw_age*, is used to hold a value for "password aging" on some systems; "password aging" is not supported on Sun systems. As such, it is effectively not used.

FILES

/etc/passwd
/etc/yp/*domainname*/passwd.byname
/etc/yp/*domainname*/passwd.byuid

SEE ALSO

getlogin(3), getgrent(3), passwd(5), ypserv(8), getpwent(3)

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

nice – change priority of a process

SYNOPSIS

nice(incr)

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs undue impact on system performance.

Negative increments are illegal, except when specified by the super-user. The priority is limited to the range –20 (most urgent) to 19 (least). Requests for values above or below these limits result in the scheduling priority being set to the corresponding limit.

The priority of a process is passed to a child process by *fork*(2).

RETURN VALUE

Upon successful completion, *nice* returns the new scheduling priority. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

The priority is not changed if:

EPERM The value of *incr* specified was negative, or greater than 40, and the effective user ID is not super-user.

SEE ALSO

nice(1), getpriority(2), setpriority(2), fork(2), renice(8)

NAME

printf, fprintf, sprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf(s, format [ , arg ] ... )
char *s, *format;

#include <varargs.h>
int _doprnt(format, args, stream)
char *format;
va_list *args;
FILE *stream;
```

DESCRIPTION

printf places output on the standard output stream **stdout**. *fprintf* places output on the named output *stream*. *sprintf* places "output", followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. *printf*, *fprintf* and *sprintf* return the number of characters transmitted (excluding the null character in the case of *sprintf*).

If an output error is encountered *printf*, *fprintf* and *sprintf* return EOF.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** (ell) specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. A **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, unsigned decimal, or unsigned hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd" where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e,E The float or double *arg* is converted in the style "[–]d.ddde±ddd," where there is one digit before the decimal point and the number after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** or **E** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

The **e**, **E**, **f**, **g**, and **G** formats print IEEE indeterminate values (infinity or not-a-number) as "Infinity" or "Nan" respectively.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

NOTE

These routines call *_doprnt*, which is an implementation-dependent routine. Each uses the variable-length argument facilities of *varargs*(3). Although it is possible to use *_doprnt* to take a list of arguments and pass them on to a routine like *printf*, not all implementations have such a routine. We strongly recommend that you use the routines described in *vprintf (3S)* instead.

SEE ALSO

putc(3S), scanf(3V), ecvt(3), printf(3V)

BUGS

Very wide fields (>128 characters) fail.

The values ''Infinity'' and ''Nan'' cannot be read by *scanf*(3V).

NAME

rand, srand – simple random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand()
```

DESCRIPTION

rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *drand48*(3) and *random*(3) provide much better, though more elaborate, random-number generators.

SEE ALSO

drand48(3), random(3), rand(3C)

BUGS

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

NAME

scanf, fscanf, sscanf – formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf(format [, pointer ] ... )
char *format;

fscanf(stream, format [, pointer ] ... )
FILE *stream;
char *format;

sscanf(s, format [, pointer ] ... )
char *s, *format;
```

DESCRIPTION

scanf reads from the standard input stream **stdin**. *fscanf* reads from the named input *stream*. *sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, or new-lines) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion characters are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be a integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e** followed by an optional +, –, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added automatically. The input field is terminated by a white space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

[indicates string data; the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be capitalized or preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

If the input ends before the first conflict or conversion, **EOF** is returned. If the input ends after the first conflict or conversion, the number of successfully matched items is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3V) strtod(3), strtol(3), scanf(3S)

DIAGNOSTICS

These functions return **EOF** on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

scanf cannot read the strings which *printf*(3V) generates for IEEE indeterminate floating point values.

scanf provides no way to convert a number in any arbitrary base (decimal, hex or octal) based on the traditional C conventions (leading 0 or 0x).

NAME

setbuf, setbuffer, setlinebuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

setlinebuf(stream)
FILE *stream;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. *fflush* (see *fclose*(3S)) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from *malloc*(3) upon the first *getc* or *putc*(3S) on the file.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

setbuf can be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered. A manifest constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

setbuffer, an alternate form of *setbuf*, can be used after a stream has been opened but before it is read or written. It causes the character array *buf* whose size is determined by the *size* argument to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer, input/output will be completely unbuffered.

setvbuf can be used after a stream has been opened but before it is read or written. *type* determines how *stream* will be buffered. Legal values for *type* (defined in **<stdio.h>**) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered. If *buf* is not the NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used.

setlinebuf is used to change the buffering on a stream from block buffered or unbuffered to line buffered. Unlike *setbuf*, *setbuffer*, and *setvbuf*, it can be used at any time that the file descriptor is active.

A file can be changed from unbuffered or line buffered to block buffered by using *freopen* (see *fopen*(3S)). A file can be changed from block buffered or line buffered to unbuffered by using *freopen* followed by *setbuf* with a buffer argument of NULL.

SEE ALSO

fopen(3V), getc(3S), putc(3S), malloc(3), fclose(3S), puts(3S), printf(3V), fread(3V), setbuf(3S)

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an ''automatic'' variable in a code block, and then failing to close the stream in the same block.

NAME

setuid – set user ID

SYNOPSIS

setuid(uid)

DESCRIPTION

setuid is used to set the real user ID and effective user ID of the calling process.

If the effective user ID of the calling process is super-user, the real user ID and effective user ID are set to *uid*.

If the effective user ID of the calling process is not super-user, but its real user ID is equal to *uid*, the effective user ID is set to *uid*.

If the effective user ID of the calling process is not super-user, but the saved set-user ID from *execve* (2) is equal to *uid*, the effective user ID is set to *uid*.

SEE ALSO

setreuid(2), getuid(2)

DIAGNOSTICS

Zero is returned if the user ID is set; –1 is returned otherwise, with the global variable *errno* set as for *setreuid*.

NAME

signal – simplified software signal facilities

SYNOPSIS

```
#include <signal.h>

(*signal(sig, func))()
int (*func)();
```

DESCRIPTION

signal is a simplified interface to the more general *sigvec*(2) facility. Programs that use *signal* in preference to *sigvec* are more likely to be portable to all UNIX systems.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the *signal* call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (other than A-line or F-line op code)
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT trap (not generated on Suns)
SIGEMT	7*	EMT trap (A-line or F-line op code)
SIGFPE	8*	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop (cannot be blocked)
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see *fcntl*(2))
SIGXCPU	24	cpu time limit exceeded (see *setrlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *setrlimit*(2))
SIGVTALRM	26	virtual time alarm (see *setitimer*(2))
SIGPROF	27	profiling timer alarm (see *setitimer*(2))
SIGWINCH	28•	window changed (see *win*(4S))
SIGLOST	29*	resource lost (see *lockd*(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs *func* is called. The value of *func* for the caught signal is reset to SIG_DFL before *func* is called, unless the signal is **SIGILL** or **SIGTRAP**

A return from the function continues the process at the point it was interrupted.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is interrupted. In particular this can occur during a *read* or *write*(2V) on a slow device (such as a terminal; but not a file) and during a *wait*(2). After the signal catching function returns, the interrupted system call may return a −1 to the calling process with *errno* set to EINTR.

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) or *vfork*(2) the child inherits all signals. An *execve*(2) resets all caught signals to the default action; ignored signals remain ignored.

NOTES

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. *Code* is a parameter of certain signals that provides additional detail. *scp* is a pointer to the *sigcontext* structure (defined in **<signal.h>**), used to restore the context from before the signal.

CODES

The following defines the codes for signals which produce them. All of these symbols are defined in **<signal.h>**:

Hardware condition	Signal	Code
Illegal instruction	SIGILL	ILL_INSTR_FAULT
Privilege violation	SIGILL	ILL_PRIVVIO_FAULT
Coprocessor protocol error	SIGILL	ILL_INSTR_FAULT
Trap #*n* (1 <= *n* <= 14)	SIGILL	ILL_TRAP_FAULT
A-line op code	SIGEMT	EMT_EMU1010
F-line op code	SIGEMT	EMT_EMU1111
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
CHK or CHK2 instruction	SIGFPE	FPE_CHKINST_TRAP
TRAPV or TRAPcc or cpTRAPcc	SIGFPE	FPE_TRAPV_TRAP
IEEE floating point compare unordered	SIGFPE	FPE_FLTBSUN_TRAP
IEEE floating point inexact	SIGFPE	FPE_FLTINEX_TRAP
IEEE floating point division by zero	SIGFPE	FPE_FLTDIV_TRAP
IEEE floating point underflow	SIGFPE	FPE_FLTUND_TRAP
IEEE floating point operand error	SIGFPE	FPE_FLTOPERR_TRAP
IEEE floating point overflow	SIGFPE	FPE_FLTOVF_FAULT
IEEE floating point signaling NaN	SIGFPE	FPE_FLTNAN_TRAP

RETURN VALUE

The previous action is returned on a successful call. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

signal will fail and no action will take place if one of the following occur:

EINVAL	*sig* is not a valid signal number.
EINVAL	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.
EINVAL	An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), ptrace(2), kill(2), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2), sigstack(2), setjmp(3), tty(4)

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep(seconds)
unsigned seconds;

DESCRIPTION

sleep suspends the current process from execution for the number of seconds specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wake-ups occur at fixed 1-second intervals and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be an arbitrary amount longer than requested because of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

sleep is implemented by setting an interval timer and pausing until it expires. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous value of the timer, the process sleeps only until the timer would have expired, and the signal which occurs with the expiration of the timer is sent one second later.

SEE ALSO

setitimer(2), sigpause(2), usleep(3)

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>

long times(buffer)
struct tms *buffer;
```

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

This is the structure returned by *times*:

```
struct tms {
        time_t  tms_utime;          /* user time */
        time_t  tms_stime;          /* system time */
        time_t  tms_cutime;         /* user time, children */
        time_t  tms_cstime;         /* system time, children */
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*.

tms_utime is the CPU time used while executing instructions in the user space of the calling process.

tms_stime is the CPU time used by the system on behalf of the calling process.

tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past. This point does not change from one invocation of *times* to another within the same process. If *times* fails, a −1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(1V), getrusage(2), wait3(2), time(3C)

NAME

ttyslot – find the slot in the utmp file of the current process

SYNOPSIS

ttyslot()

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/ttys** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/ttys

DIAGNOSTICS

A value of −1 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

intro – introduction to other libraries

DESCRIPTION

This section contains manual pages describing other libraries, which are available only from C. The list below includes libraries which provide device independent plotting functions, terminal independent screen management routines for two dimensional non-bitmap display terminals, and functions for managing data bases with inverted indexes. All functions are located in separate libraries indicated in each manual entry.

FILES

/usr/lib/libcurses.a	screen management routines (see *curses*(3X))
/usr/lib/libdbm.a	data base management routines (see *dbm*(3X))
/usr/lib/libmp.a	multiple precision math library (see *mp*(3X))
/usr/lib/libplot.a	plot routines (see *plot*(3X))
/usr/lib/lib300.a	"
/usr/lib/lib300s.a	"
/usr/lib/lib450.a	"
/usr/lib/lib4014.a	"
/usr/lib/libtermcap.a	terminal handling routines (see *termcap*(3X))
/usr/lib/libtermcap_p.a	
/usr/lib/libtermlib.a	(link to /usr/lib/libtermcap.a)
/usr/lib/libtermlib_p.a	(link to /usr/lib/libtermcap_p.a)
/usr/lib/libresolv.a	Internet server routines (see *resolver*(3X))

NAME

curses – screen functions with "optimal" cursor motion

SYNOPSIS

cc [*flags*] *files* **–lcurses –ltermcap** [*libraries*]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

ioctl(2), getenv(3), tty(4), termcap(5)
Programmer's Reference Manual for Curses

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
cbreak()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
flusok(win,boolf)	set flush-on-refresh flag for *win*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocbreak()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*

resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchline(win,y,sx,ex)	mark line *y* *sx* through *sy* as changed
touchoverlap(win1,win2)	mark overlap of *win1* on *win2* as changed
touchwin(win)	''change'' all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert character into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

NAME

dbm, dbminit, fetch, store, delete, firstkey, nextkey – data base subroutines

SYNOPSIS

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;

dbmclose()
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm**.

*Key*s and *content*s are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

A database may be closed by calling *dbmclose*. You must close a database before opening a new one.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

NAME

mp, itom, madd, msub, mult, mdiv, min, mout, pow, gcd, rpow, xtom, mtox, mfree – multiple precision integer arithmetic

SYNOPSIS

```
#include <mp.h>

madd(a, b, c)
MINT *a, *b, *c;

msub(a, b, c)
MINT *a, *b, *c;

mult(a, b, c)
MINT *a, *b, *c;

mdiv(a, b, q, r)
MINT *a, *b, *q, *r;

min(a)
MINT *a;

mout(a)
MINT *a;

pow(a, b, c, d)
MINT *a, *b, *c, *d;

gcd(a, b, c)
MINT *a, *b, *c;

rpow(a, n, b)
MINT *a, *b;
short n;

msqrt(a, b, r)
MINT *a, *b, *r;

sdiv(a, n, q, r)
MINT *a, *q;
short n, *r;

MINT *itom(n)
short n;

MINT *xtom(s)
char *s;

char *mtox(a)
MINT *a;

void mfree(a)
MINT *a;
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type *MINT*. Pointers to a *MINT* should be initialized using the function *itom*, which sets the initial value to *n*. Alternatively, *xtom* may be used to initialize a *MINT* from a string of hexadecimal digits. *mfree* may be used to release the storage allocated by these routines.

Madd, *msub* and *mult* assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. *Mdiv* assigns the quotient and remainder, respectively, to its third and fourth arguments. *Sdiv* is like *mdiv* except that the divisor is an ordinary integer. *Msqrt* produces the square root and

remainder of its first argument. *Rpow* calculates *a* raised to the power *b*, while *pow* calculates this reduced modulo *m*. *Min* and *mout* do decimal input and output. *mtox* provides the inverse of *xtom*.

Use the **–lmp** loader option to obtain access to these functions.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

FILES

/usr/lib/libmp.a

NAME

ndbm, dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

SYNOPSIS

```
#include <ndbm.h>

typedef struct {
  char *dptr;
  int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
  char *file;
  int flags, mode;

dbm_close(db)
  DBM *db;

datum dbm_fetch(db, key)
  DBM *db;
  struct key;
  datum key;

dbm_store(db, key, content, flags)
  DBM *db;
  datum key, content;
  int flags;

dbm_delete(db, key)
  DBM *db;
  datum key;

datum dbm_firstkey(db)
  DBM *db;

datum dbm_nextkey(db)
  DBM *db;

datum dbm_error(db)
  DBM *db;

datum dbm_clearerr(db)
  DBM *db;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses.

*key*s and *content*s are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbm_open*. This will open and/or create the files *file*.**dir** and *file*.**pag** depending on the flags parameter (see *open*(2V)).

Once open, the data stored under a key is accessed by *dbm_fetch* and data is placed under a key by *dbm_store*. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE. DBM_INSERT** will only insert new entries into the database and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by *dbm_delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *dbm_firstkey* and *dbm_nextkey*. *dbm_firstkey* will return the first key in the

database. *dbm_nextkey* will return the next key in the database. This code will traverse the data base:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

dbm_error returns non-zero when an error has occured reading or writing the database. *dbm_clearerr* Resets the error condition on the named database.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. *dbm_store* will return an error in the event that a disk block fills with inseparable data.

dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *dbm_firstkey* and *dbm_nextkey* depends on a hashing function, not on anything interesting.

NAME

plot, openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl – graphics interface

SYNOPSIS

openpl()

erase()

label(s)
char s[];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[];

space(x0, y0, x1, y1)

closepl()

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot*(5) for a description of their effect. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

String arguments to *label* and *linemod* are null-terminated, and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following *ld*(1) options:

–lplot device-independent graphics stream on standard output for *plot*(1G) filters
–l300 GSI 300 terminal
–l300s GSI 300S terminal
–l450 GSI 450 terminal
–l4014 Tektronix 4014 terminal
–lplotaed
AED 512 color graphics terminal
–lplotbg BBN bitgraph graphics terminal
–lplotdumb
Dumb terminals without cursor addressing or line printers
–lplotgigi
DEC Gigi terminals
–lplot2648
Hewlett Packard 2648 graphics terminal
–lplot7221
Hewlett Packard 7221 graphics terminal
–lplotimagen
Imagen laser printer (default 240 dots-per-inch resolution).

SEE ALSO

plot(5), plot(1G), graph(1G)

FILES

/usr/lib/libplot.a
/usr/lib/lib300.a
/usr/lib/lib300s.a
/usr/lib/lib450.a
/usr/lib/lib4014.a
/usr/lib/libplotaed.a
/usr/lib/libplotbg.a
/usr/lib/libplotdumb.a
/usr/lib/libplotgigi.a
/usr/lib/libplot2648.a
/usr/lib/libplot7221.a
/usr/lib/libplotimagen.a

NAME

termcap, tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). These are low level routines; see *curses*(3X) for a higher level package.

Tgetent extracts the entry for terminal *name* into the *bp* buffer, with the current size of the tty (usually a window). This allows pre-SunWindows programs to run in a window of arbitrary size. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns –1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than */etc/termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file */etc/termcap*. Note that if the window size changes, the "lines" and "columns"entries in *bp* are no longer correct. See the *Sunwindows Reference Manual* for details regarding [how to handle] this.

Tgetnum gets the numeric value of capability *id*, returning –1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information. *Tgetstr* returns the string pointer if successful. Otherwise it returns zero.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since *tgoto* may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then *tgoto* returns "OOPS".

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the encoded output speed of the terminal as described in *tty*(4). The external variable PC should contain a pad character to be used (from the pc capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	–ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3X), tty(4), termcap(5)

NAME

intro – introduction to special files and hardware support

DESCRIPTION

This section describes device interfaces (drivers) in the operating system for disks, tapes, serial communications, high-speed network communications, and other devices such as mice, frame buffers and windows. The section is divided into a few subsections:

- Sun-specific drivers are grouped in '4S'.
- Protocol families are grouped in '4F'.
- Protocols and raw interfaces are treated in '4P'.
- Network interfaces are grouped in '4N'.

The operating system can be built with or without many of the drivers listed here. For most of them, the SYNOPSIS section of the manual page gives the syntax of the line to include in a kernel configuration file if you wish to include the driver in a system. See *config*(8) for a description of this process.

Several manual pages will contain SYNOPSIS sections specific to the Sun-2 and Sun-3 architectures. Where a SYNOPSIS section appears without any specific architecture against it, it applies to both the Sun-2 and Sun-3 architectures. Where a SYNOPSIS section appears with only one specific architecture against it, it applies only to that specific architecture.

The pages for most drivers also include a DIAGNOSTICS section listing error messages the driver may produce. These messages appear on the system console, and also in the system error log file */usr/adm/messages*.

DEVICES ALWAYS PRESENT

Drivers which are present in every kernel include a driver for the paging device, *drum*(4); drivers for accessing physical, virtual, and I/O space, *mem*(4S); and drivers for the data sink, *null*(4).

COMMUNICATIONS DEVICES

Communications lines are most often used with the terminal driver described in *tty*(4). The terminal driver runs on communications lines provided either by a communications driver such as *mti*(4S) or *zs*(4S) or by a virtual terminal. The virtual terminal may be provided either by the Sun console monitor, *cons*(4S), or by a true pseudo-terminal, *pty*(4), used in applications such as windowing or remote networking.

MAGNETIC TAPE DEVICES

Magnetic tapes all provide the interface described in *mtio*(4). Tape devices for the Sun include *ar*(4S), *tm*(4S), *st*(4S), and *xt*(4S).

DISK DEVICES

Disk controllers provide standard block and raw interfaces, as well as a set of ioctl's defined in *dkio*(4S), which support getting and setting disk geometry and partition information. Drivers available for the Sun include *xy*(4S), *ip*(4S), and *sd*(4S).

PROTOCOL FAMILIES

The operating system supports one or more protocol families for local network communications. The only complete protocol family in this version of the system is the Internet protocol family; see *inet*(4F). Each protocol family provides basic services — packet fragmentation and reassembly, routing, addressing, and basic transport — to each protocol implementation. A protocol family is normally composed of a number of protocols, one per *socket*(2) type. A protocol family is not required to support all socket types.

The primary network support is for the Internet protocol family described in *inet*(4F). Major protocols in this family include the Internet Protocol, *ip*(4P), describing the universal datagram format, the stream Transmission Control Protocol *tcp*(4P), the User Datagram Protocol *udp*(4P), the Address Resolution Protocol *arp*(4P), the Internet Control Message Protocol *icmp*(4P), and the Network Interface Tap *nit*(4P). The primary network interface is for the 10 Megabit Ethernet; see *ec*(4S), *ie*(4S), and *le*(4S). A software loopback interface, *lo*(4) also exists. General properties of these (and all) network interfaces are described in *if*(4N).

The general support in the system for local network routing is described in *routing*(4N); these facilities apply to all protocol families.

MISCELLANEOUS DEVICES

Miscellaneous devices include color frame buffers *cg**(4S), monochrome frame buffers *bw**(4S), the console frame buffer *fb*(4S), the graphics processor interface *gpone*(4S), the console mouse *mouse*(4S), and the window devices *win*(4S).

GENERAL IOCTL CALLS

In general, ioctl calls relating to a specific device are mentioned with the description for that device. There are however a bunch of ioctl calls that apply to files in general. These are described here. The form of the ioctl call for file control is:

```
#include <sys/ioctl.h>
ioctl(fd, request, argp)
int fd, request;
int *argp;
```

FIOCLEX — Set set close-on-exec flag for the file descriptor specified by **fd**. This flag is also manipulated by the F_SETFD command of *fcntl*(2). The **argp** argument is not used in this call.

FIONCLEX — Remove close-on-exec flag for the file descriptor specified by **fd**. The **argp** argument is not used in this call.

FIONREAD — Returns in the long integer whose address is **argp** the number of immediately readable characters from whatever the descriptor specified by **fd**. refers to. This works for files, pipes, and terminals.

FIONBIO — Set or clear non-blocking I/O. If the value pointed to by **argp** is a 1 (one) the descriptor is set for non-blocking I/O. If the value pointed to by **argp** is a 0 (zero) the descriptor is cleared for non-blocking I/O.

FIOASYNC — Set or clear asynchronous I/O. If the value pointed to by **argp** is a 1 (one) the descriptor is set for asynchronous I/O. If the value pointed to by **argp** is a 0 (zero) the descriptor is cleared for asynchronous I/O.

FIOSETOWN — Set the process-group ID that will subsequently receive SIGIO or SIGURG signals for this descriptor.

FIOGETOWN — Get the process-group ID that is receiving SIGIO or SIGURG signals for this descriptor.

SEE ALSO

fcntl(2)

NAME

ar – Archive 1/4 inch Streaming Tape Drive

SYNOPSIS — SUN-2

device ar0 at mbio ? csr 0x200 priority 3
device ar1 at mbio ? csr 0x208 priority 3

DESCRIPTION

The Archive tape controller is a Sun 'QIC-II' interface to an Archive streaming tape drive. It provides a standard tape interface to the device, see *mtio*(4), with some deficiencies listed under BUGS below.

The maximum blocksize for the raw device is limited only by available memory.

FILES

/dev/rar*
/dev/nrar* non-rewinding

SEE ALSO

mtio(4)

DIAGNOSTICS

ar*: would not initialize.
"ar*: already open."
The tape can be open by only one process at a time.
ar*: no such drive.
ar*: no cartridge in drive.
ar*: cartridge is write protected.
ar: interrupt from unitialized controller %x.
ar*: many retries, consider retiring this tape.
ar*: %b error at block # %d punted.
ar*: %b error at block # %d.
ar: giving up on Rdy, try again.

BUGS

The tape cannot reverse direction so the BSF and BSR ioctls are not supported.
The FSR ioctl is not supported.
The system will hang if the tape is removed while running.
When using the raw device, the number of bytes in any given transfer must be a multiple of 512 bytes. If it is not, the device driver returns an error.
The driver will only write an end of file mark on close if the last operation was a write, without regard for the mode used when opening the file. This will cause empty files to be deleted on a raw tape copy operation.

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending messages are transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently ''transmitted'' packet is kept.

To enable communications with systems which do not use ARP, ioctls are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
        struct sockaddr   arp_pa;         /* protocol address */
        struct sockaddr   arp_ha;         /* hardware address */
        int       arp_flags;              /* flags */
};
/* arp_flags field values */
#define ATF_COM             2       /* completed entry (arp_ha valid) */
#define  ATF_PERM           4       /* permanent entry */
#define  ATF_PUBL           8       /* publish (respond for other host) */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM and ATF_PUBL. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 4 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a Sun to act as an "ARP server" which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP watches passively for hosts impersonating the local host (that is, a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

SEE ALSO

ec(4S), ie(4S), inet(4F), arp(8C), ifconfig(8C)
An Ethernet Address Resolution Protocol, RFC826, Dave Plummer, MIT (Sun 800-1059-01)

BUGS

ARP packets on the Ethernet use only 42 bytes of data, however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NAME

bk – line discipline for machine-machine communication

SYNOPSIS

pseudo-device bk

DESCRIPTION

This line discipline provides a replacement for the tty driver *tty*(4) when high speed output to and especially input from another machine is to be transmitted over an asynchronous communications line. The discipline was designed for use by a (now obsolete) store-and-forward local network running over serial lines. It may be suitable for uploading of data from microprocessors into the system. If you are going to send data over asynchronous communications lines at high speed into the system, you must use this discipline, as the system otherwise may detect high input data rates on terminal lines and disable the lines; in any case the processing of such data when normal terminal mechanisms are involved saturates the system.

The line discipline is enabled by a sequence:

```
#include <sgtty.h>
int ldisc = NETLDISC, fildes; ...
ioctl(fildes, TIOCSETD, &ldisc);
```

A typical application program then reads a sequence of lines from the terminal port, checking header and sequencing information on each line and acknowledging receipt of each line to the sender, who then transmits another line of data. Typically several hundred bytes of data and a smaller amount of control information will be received on each handshake.

The old standard teletype discipline can be restored by doing:

```
ldisc = OTTYDISC;
ioctl(fildes, TIOCSETD, &ldisc);
```

While in networked mode, normal teletype output functions take place. Thus, if an 8 bit output data path is desired, it is necessary to prepare the output line by putting it into RAW mode using *ioctl*(2). This must be done **before** changing the discipline with TIOCSETD, as most *ioctl*(2) calls are disabled while in network line-discipline mode.

When in network mode, input processing is very limited to reduce overhead. Currently the input path is only 7 bits wide, with newline the only character terminating an input record. Each input record must be read and acknowledged before the next input is read as the system refuses to accept any new data when there is a record in the buffer. The buffer is limited in length, but the system guarantees to always be willing to accept input resulting in 512 data characters and then the terminating newline.

User level programs should provide sequencing and checksums on the information to guarantee accurate data transfer.

SEE ALSO

tty(4)

NAME

bwone – Sun-1 black and white frame buffer

SYNOPSIS — SUN-2

device bwone0 at mbmem ? csr 0xc0000 priority 3

DESCRIPTION

The *bwone* interface provides access to Sun-1 black and white graphics controller boards. It supports the FBIOGTYPE ioctl which programs can use to determine the characteristics of the display device; see *fbio*(4S).

bwone also supports the FBIOGPIXRECT ioctl which allows SunWindows to be run on it; see *fbio*(4S).

Reading or writing to the frame buffer is not allowed – you must use the *mmap*(2) system call to map the board into your address space.

FILES

/dev/bwone[0-9]

SEE ALSO

mmap(2), fb(4S), fbio(4S)

BUGS

Use of vertical-retrace interrupts is not supported.

The FBVIDEO_ON value returned by the FBIOGVIDEO ioctl may be incorrect. See *fbio*(4S).

NAME

bwtwo – Sun-3/Sun-2 black and white frame buffer

SYNOPSIS — SUN-3

device bwtwo0 at obmem 1 csr 0xff000000 priority 4
device bwtwo0 at obmem 2 csr 0x100000 priority 4
device bwtwo0 at obmem 3 csr 0xff000000 priority 4
device bwtwo0 at obmem 4 csr 0xff000000

The first synopsis line given above should be used to generate a kernel for a Sun-3/160; the second, for a Sun-3/75M; the third, for a Sun-3/260; and the fourth, for a Sun-3/110.

SYNOPSIS — SUN-2

device bwtwo0 at obmem 1 csr 0x700000 priority 4
device bwtwo0 at obio 2 csr 0x0 priority 4

The first synopsis line given above should be used to generate a kernel for a Sun-2/120 or Sun-2/170; the second, for a Sun-2/50 or Sun-2/160.

DESCRIPTION

The *bwtwo* interface provides access to Sun Monochrome Video Controller boards.

bwtwo supports the FBIOGTYPE ioctl, which may be used to determine the characteristics of the display device, and the FBIOGPIXRECT ioctl, which allows SunWindows to be run on it (see *fbio*(4S)).

If **flags 0x1** is specified, frame buffer write operations are buffered through regular high-speed RAM. This "copy memory" mode of operation speeds the write operations, but consumes an extra 128K bytes of memory.

Reading or writing to the frame buffer is not allowed — you must use the *mmap*(2) system call to map the board into your address space.

FILES

/dev/bwtwo[0-9]

SEE ALSO

mmap(2), fb(4S), fbio(4S), cgfour(4S)

BUGS

Use of vertical-retrace interrupts is not supported.

The FBVIDEO_ON value returned by the FBIOGVIDEO ioctl may be incorrect. See *fbio*(4S).

NAME

cgfour – Sun-3 color graphics interface

SYNOPSIS — SUN-3

cgfour0 at obmem 4 csr 0xff000000

DESCRIPTION

The *cgfour* is the Sun-3/110 color frame buffer, normally supplied with a 19'' color, 19'' grayscale, or 15'' color 66 Hz non-interlaced color monitor. It provides the standard frame buffer interface as defined in *fbio*(4S).

In addition to the ioctls described under *fbio*(4s), the *cgfour* interface responds to two *cgfour*-specific colormap ioctls, FBIOPUTCMAP and FBIOGETCMAP. FBIOPUTCMAP returns no information other than success/failure via the ioctl return value. FBIOGETCMAP returns its information in the arrays pointed to by the red, green, and blue members of its **fbcmap** structure argument; **fbcmap** is defined in *<sun/fbio.h>* as:

```
struct  fbcmap {
        int             index;          /* first element (0 origin) */
        int             count;          /* number of elements */
        unsigned char   *red;           /* red color map elements */
        unsigned char   *green;         /* green color map elements */
        unsigned char   *blue;          /* blue color map elements */
};
```

The driver uses color board vertical-retrace interrupts to load the colormap.

Currently the ioctls FBIOSATTR and FBIOGATTR are only supported by the *cgfour* frame buffer. See *fbio*(4S).

FILES

/dev/cgfour0

SEE ALSO

mmap(2), fbio(4S)

Sun-3/1xx CPU Board Hardware Engineering Manual

BUGS

The FBVIDEO_ON value returned by the FBIOGVIDEO ioctl may be incorrect. See *fbio*(4S).

NAME

cgone – Sun-1 color graphics interface

SYNOPSIS — SUN-2

device cgone0 at mbmem ? csr 0xec000 priority 3

DESCRIPTION

The *cgone* interface provides access to the Sun-1 color graphics controller board, which is normally supplied with a 13" or 19" RS170 color monitor. It provides the standard frame buffer interface as defined in *fbio*(4S).

It supports the FBIOGPIXRECT ioctl which allows SunWindows to be run on it; see *fbio*(4S)

The hardware consumes 16 kilobytes of Multibus memory space. The board starts at standard addresses 0xE8000 or 0xEC000. The board must be configured for interrupt level 3.

FILES

/dev/cgone[0-9]

SEE ALSO

mmap(2), fbio(4S)

BUGS

Use of color board vertical-retrace interrupts is not supported.

NAME

cgtwo – Sun-3/Sun-2 color graphics interface

SYNOPSIS — SUN-3

cgtwo0 at vme24d16 ? csr 0x400000

SYNOPSIS — SUN-2

cgtwo0 at vme24 ? csr 0x400000

DESCRIPTION

The *cgtwo* interface provides access to the Sun-3/Sun-2 color graphics controller board, which is normally supplied with a 19'' 66 Hz non-interlaced color monitor. It provides the standard frame buffer interface as defined in *fbio*(4S).

The hardware consumes 4 megabytes of VME bus address space. The board starts at standard address 0x400000. The board must be configured for interrupt level 3.

FILES

/dev/cgtwo[0-9]

SEE ALSO

mmap(2), fbio(4S)

NAME

console – console driver and terminal emulator for the Sun workstation

SYNOPSIS

None; included in standard system.

DESCRIPTION

Cons is an indirect driver for the Sun workstation console, which implements a standard UNIX system terminal. *Cons* is implemented by calling the PROM resident monitor or other kernel UART drivers (*zs*(4S)) to perform I/O to and from the current system console, which is either a Sun frame buffer or an RS232 port.

When the Sun window system *win*(4S) is active, console input is directed through the window system rather than being read from the hardware console.

An ioctl TIOCCONS can be applied to serial devices other than the console to route output which would normally appear on the console to the other devices instead. Thus, the window system does a TIOCCONS on a pseudoterminal to route console output to the pseudoterminal rather than routing output through the PROM monitor to the screen, since routing output through the PROM monitor destroys the integrity of the screen. Note however, that when you use TIOCCONS in this way, the console *input* is routed from the pseudoterminal as well.

ANSI STANDARD TERMINAL EMULATION

The Sun Workstation's PROM monitor provides routines that emulates a standard ANSI X3.64 terminal.

Note that the VT100 also follows the ANSI X3.64 standard but both the Sun and the VT100 have nonstandard extensions to the ANSI X3.64 standard. The Sun terminal emulator and the VT100 are *not* compatible in any true sense.

The Sun console displays 34 lines of 80 ASCII characters per line, with scrolling, (x, y) cursor addressability, and a number of other control functions.

The Sun console displays a non-blinking block cursor which marks the current line and character position on the screen. ASCII characters between 0x20 (space) and 0x7E (tilde) inclusive are printing characters — when one is written to the Sun console (and is not part of an escape sequence), it is displayed at the current cursor position and the cursor moves one position to the right on the current line. If the cursor is already at the right edge of the screen, it moves to the first character position on the next line. If the cursor is already at the right edge of the screen on the bottom line, the Line-feed function is performed (see control–J below), which scrolls the screen up by one or more lines or wraps around, before moving the cursor to the first character position on the next line.

Control Sequence Syntax

The Sun console defines a number of control sequences which may occur in its input. When such a sequence is written to the Sun console, it is not displayed on the screen, but effects some control function as described below, for example, moves the cursor or sets a display mode.

Some of the control sequences consist of a single character. The notation

control–*X*

for some character *X* , represents a control character.

Other ANSI control sequences are of the form

ESC [<*params*> <*char*>

Spaces are included only for readability; these characters must occur in the given sequence without the intervening spaces.

ESC represents the ASCII escape character (ESC, control–[, 0x1B).

[The next character is a left square bracket '[' (0x5B).

<*params*>
are a sequence of zero or more decimal numbers made up of digits between 0 and 9, separated by semicolons.

<*char*> represents a function character, which is different for each control sequence.

Some examples of syntactically valid escape sequences are (again, ESC represent the single ASCII character 'Escape'):

ESC[m	*select graphic rendition with default parameter*
ESC[7m	*select graphic rendition with reverse image*
ESC[33;54H	*set cursor position*
ESC[123;456;0;;3;B	*move cursor down*

Syntactically valid ANSI escape sequences which are not currently interpreted by the Sun console are ignored. Control characters which are not currently interpreted by the Sun console are also ignored.

Each control function requires a specified number of parameters, as noted below. If fewer parameters are supplied, the remaining parameters default to 1, except as noted in the descriptions below.

If more than the required number of parameters is supplied, only the last *n* are used, where *n* is the number required by that particular command character. Also, parameters which are omitted or set to zero are reset to the default value of 1 (except as noted below).

Consider, for example, the command character M which requires one parameter. ESC[;M and ESC[0M and ESC[M and ESC[23;15;32;1M are all equivalent to ESC[1M and provide a parameter value of 1. Note that ESC[;5M (interpreted as 'ESC[5M') is *not* equivalent to ESC[5;M (interpreted as 'ESC[5;1M') which is ultimately interpreted as 'ESC[1M').

In the syntax descriptions below, parameters are represented as '#' or '#1;#2'.

ANSI Control Functions

The following paragraphs specify the ANSI control functions implemented by the Sun console. Each description gives:

- the control sequence syntax
- the hex equivalent of control characters where applicable
- the control function name and ANSI or Sun abbreviation (if any).
- description of parameters required, if any
- description of the control function
- for functions which set a mode, the initial setting of the mode. The initial settings can be restored with the SUNRESET escape sequence.

Control Character Functions

control–G (0x7) Bell (BEL)

The Sun Workstation Model 100 and 100U is not equipped with an audible bell. It 'rings the bell' by flashing the entire screen. The Sun-2 models have an audible bell which beeps. The window system flashes the window.

control–H (0x8) Backspace (BS)

The cursor moves one position to the left on the current line. If it is already at the left edge of the screen, nothing happens.

control–I (0x9) Tab (TAB)

The cursor moves right on the current line to the next tab stop. The tab stops are fixed at every multiple of 8 columns. If the cursor is already at the right edge of the screen, nothing happens; otherwise the cursor moves right a minimum of one and a maximum of eight character positions.

control–J (0xA) Line-feed (LF)

The cursor moves down one line, remaining at the same character position on the line. If the cursor is already at the bottom line, the screen either scrolls up or 'wraps around' depending on the setting of an internal variable *S* (initially 1) which can be changed by the ESC[r control sequence.

If *S* is greater than zero, the entire screen (including the cursor) is scrolled up by *S* lines before executing the Line-feed. The top *S* lines scroll off the screen and are lost. *S* new blank lines scroll onto the bottom of the screen. After scrolling, the line-feed is executed by moving the cursor down one line.

If *S* is zero, 'wrap-around' mode is entered. 'ESC [1 r' exits back to scroll mode. If a linefeed occurs on the bottom line in wrap mode, the cursor goes to the same character position in the top line of the screen. When any linefeed occurs, the line that the cursor moves to is cleared. This means that no scrolling occurs. Wrap-around mode is not implemented in the window system.

The screen scrolls as fast as possible depending on how much data is backed up awaiting printing. Whenever a scroll must take place and the console is in normal scroll mode ('ESC [1 r'), it scans the rest of the data awaiting printing to see how many linefeeds occur in it. This scan stops when any control character from the set {VT, FF, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FS, GS, RS, US} is found. At that point, the screen is scrolled by N lines (N at least 1) and processing continues. The scanned text is still processed normally to fill in the newly created lines. This results in much faster scrolling with scrolling as long as no escape codes or other control characters are intermixed with the text.

See also the discussion of the 'Set scrolling' (ESC[r) control funtion below.

control–K (0xB) Reverse Line-feed
The cursor moves up one line, remaining at the same character position on the line. If the cursor is already at the top line, nothing happens.

control–L (0xC) Form-feed (FF)
The cursor is postioned to the Home position (upper-left corner) and the entire screen is cleared.

control–M (0xD) Return (CR)
The cursor moves to the leftmost character position on the current line.

Escape Sequence Functions

control–[(0x1B) Escape (ESC)
This is the escape character. Escape initiates a multi-character control sequence.

ESC[#@ Insert Character (ICH)
Takes one parameter, # (default 1). Inserts # spaces at the current cursor position. The tail of the current line starting at the current cursor position inclusive is shifted to the right by # character positions to make room for the spaces. The rightmost # character positions shift off the line and are lost. The position of the cursor is unchanged.

ESC[#A Cursor Up (CUU)
Takes one parameter, # (default 1). Moves the cursor up # lines. If the cursor is fewer than # lines from the top of the screen, moves the cursor to the topmost line on the screen. The character position of the cursor on the line is unchanged.

ESC[#B Cursor Down (CUD)
Takes one parameter, # (default 1). Moves the cursor down # lines. If the cursor is fewer than # lines from the bottom of the screen, move the cursor to the last line on the screen. The character position of the cursor on the line is unchanged.

ESC[#C Cursor Forward (CUF)
Takes one parameter, # (default 1). Moves the cursor to the right by # character positions on the current line. If the cursor is fewer than # positions from the right edge of the screen, moves the cursor to the rightmost position on the current line.

ESC[#D Cursor Backward (CUB)
Takes one parameter, # (default 1). Moves the cursor to the left by # character positions on the current line. If the cursor is fewer than # positions from the left edge of the screen, moves the

cursor to the leftmost position on the current line.

ESC[#E Cursor Next Line (CNL)
Takes one parameter, # (default 1). Positions the cursor at the leftmost character position on the #-th line below the current line. If the current line is less than # lines from the bottom of the screen, postions the cursor at the leftmost character position on the bottom line.

ESC[#1;#2f Horizontal And Vertical Position (HVP)
or
ESC[#1;#2H Cursor Position (CUP)
Takes two parameters, #1 and #2 (default 1, 1). Moves the cursor to the #2-th character position on the #1-th line. Character positions are numbered from 1 at the left edge of the screen; line positions are numbered from 1 at the top of the screen. Hence, if both parameters are omitted, the default action moves the cursor to the home position (upper left corner). If only one parameter is supplied, the cursor moves to column 1 of the specified line.

ESC[J Erase in Display (ED)
Takes no parameters. Erases from the current cursor position inclusive to the end of the screen. In other words, erases from the current cursor position inclusive to the end of the current line and all lines below the current line. The cursor position is unchanged.

ESC[K Erase in Line (EL)
Takes no parameters. Erases from the current cursor position inclusive to the end of the current line. The cursor position is unchanged.

ESC[#L Insert Line (IL)
Takes one parameter, # (default 1). Makes room for # new lines starting at the current line by scrolling down by # lines the portion of the screen from the current line inclusive to the bottom. The # new lines at the cursor are filled with spaces; the bottom # lines shift off the bottom of the screen and are lost. The position of the cursor on the screen is unchanged.

ESC[#M Delete Line (DL)
Takes one parameter, # (default 1). Deletes # lines beginning with the current line. The portion of the screen from the current line inclusive to the bottom is scrolled upward by # lines. The # new lines scrolling onto the bottom of the screen are filled with spaces; the # old lines beginning at the cursor line are deleted. The position of the cursor on the screen is unchanged.

ESC[#P Delete Character (DCH)
Takes one parameter, # (default 1). Deletes # characters starting with the current cursor position. Shifts to the left by # character positions the tail of the current line from the current cursor position inclusive to the end of the line. Blanks are shifted into the rightmost # character positions. The position of the cursor on the screen is unchanged.

ESC[#m Select Graphic Rendition (SGR)
Takes one parameter, # (default 0). Note that, unlike most escape sequences, the parameter defaults to zero if omitted. Invokes the graphic rendition specified by the parameter. All following printing characters in the data stream are rendered according to the parameter until the next occurrence of this escape sequence in the data stream. Currently only two graphic renditions are defined:

0 Normal rendition.

7 Negative (reverse) image.

Negative image displays characters as white-on-black if the screen mode is currently black-on white, and vice-versa. Any non-zero value of # is currently equivalent to 7 and selects the negative image rendition.

ESC[p Black On White (SUNBOW)
Takes no parameters. Sets the screen mode to black-on-white. If the screen mode is already black-on-white, has no effect. In this mode spaces display as solid white, other characters as

black-on-white. The cursor is a solid black block. Characters displayed in negative image rendition (see 'Select Graphic Rendition' above) is white-on-black in this mode. This is the initial setting of the screen mode on reset.

ESC[q White On Black (SUNWOB)
Takes no parameters. Sets the screen mode to white-on-black. If the screen mode is already white-on-black, has no effect. In this mode spaces display as solid black, other characters as white-on-black. The cursor is a solid white block. Characters displayed in negative image rendition (see 'Select Graphic Rendition' above) is black-on-white in this mode. The initial setting of the screen mode on reset is the alternative mode, black on white.

ESC[#r Set scrolling (SUNSCRL)
Takes one parameter, # (default 0). Sets to # an internal register which determines how many lines the screen scrolls up when a line-feed function is performed with the cursor on the bottom line. A parameter of 2 or 3 introduces a small amount of 'jump' when a scroll occurs. A parameter of 34 clears the screen rather than scrolling. The initial setting is 1 on reset.

A parameter of zero initiates 'wrap mode' instead of scrolling. In wrap mode, if a linefeed occurs on the bottom line, the cursor goes to the same character position in the top line of the screen. When any linefeed occurs, the line that the cursor moves to is cleared. This means that no scrolling ever occurs. 'ESC [1 r' exits back to scroll mode.

For more information, see the description of the Line-feed (control–J) control function above.

ESC[s Reset terminal emulator (SUNRESET)
Takes no parameters. Resets all modes to default, restores current font from PROM. Screen and cursor position are unchanged.

4014 TERMINAL EMULATION

The PROM monitor for Sun models 100U and 150U provides the Sun Workstation with the capability to emulate a subset of the Tektronix 4014 terminal. This feature does not exist in Sun-2 PROMs and will be removed from models 100U and 150U in future Sun releases. *Tektool*(1) provides Tektronix 4014 terminal emulation and should be used instead of relying on the capabilities of the PROM monitor.

FILES

/dev/console	
/dev/ttya	alternate console (serial port)

SEE ALSO

kb(4S), tty(4), zs(4S), tektool(1)

ANSI Standard X3.64, 'Additional Controls for Use with ASCII', Secretariat: CBEMA, 1828 L St., N.W., Washington, D.C. 20036.

BUGS

TIOCCONS should be restricted to the owner of /dev/console.

NAME

des – DES encryption chip interface

SYNOPSIS — SUN-3

des0 at obio ? csr 0x1c0000

#include <sys/des.h>

SYNOPSIS — SUN-2

des0 at virtual ? csr 0xee1800

#include <sys/des.h>

DESCRIPTION

The *des* driver provides a high level interface to the AmZ8068 Data Ciphering Processor, a hardware implementation of the NBS Data Encryption Standard.

The high level interface provided by this driver is hardware independent and could be shared by future drivers in other systems.

The interface allows access to two modes of the DES algorithm: Electronic Code Book (ECB) and Cipher Block Chaining (CBC). All access to the DES driver is through *ioctl*(2) calls rather than through reads and writes; all encryption is done in-place in the user's buffers. The ioctls provided are:

DESIOCBLOCK

This call encrypts/decrypts an entire buffer of data, whose address and length are passed in the **struct desparams** addressed by the argument. The length must be a multiple of 8 bytes.

DESIOCQUICK

This call encrypts/decrypts a small amount of data quickly. The data is limited to DES_QUICKLEN bytes, and must be a multiple of 8 bytes. Rather than being addresses, the data is passed directly in the **struct desparams** argument.

FILES

/dev/des

SEE ALSO

des_crypt(3), des(1)

Federal Information Processing Standards Publication 46

AmZ8068 DCP Product Description, Advanced Micro Devices

NAME

dkio – generic disk control operations

DESCRIPTION

All Sun disk drivers support a set of ioctl's for disk formatting and labelling operations. Basic to these ioctl's are the definitions in <sun/dkio.h>:

```
/*
 * Structures and definitions for disk io control commands
 */

/* Disk identification */
struct dk_info {
        int     dki_ctlr;       /* controller address */
        short   dki_unit;               /* unit (slave) address */
        short   dki_ctype;              /* controller type */
        short   dki_flags;              /* flags */
};
/* controller types */
#define  DKC_UNKNOWN        0
#define  DKC_SMD2180        1
#define  DKC_DSD5215        5
#define  DKC_XY450          6
#define  DKC_ACB4000        7
#define DKC_MD21            8

/* flags */
#define  DKI_BAD144   0x01   /* use DEC std 144 bad sector fwding */
#define  DKI_MAPTRK   0x02   /* controller does track mapping */
#define  DKI_FMTTRK   0x04   /* formats only full track at a time */
#define  DKI_FMTVOL   0x08   /* formats only full volume at a time */

/* Definition of a disk's geometry */
struct dk_geom {
        unsigned short  dkg_ncyl;       /* # of data cylinders */
        unsigned short  dkg_acyl;       /* # of alternate cylinders */
        unsigned short  dkg_bcyl;       /* cyl offset (for fixed head area) */
        unsigned short  dkg_nhead;      /* # of heads */
        unsigned short  dkg_bhead;      /* head offset (for Larks, etc.) */
        unsigned short  dkg_nsect;      /* # of sectors per track */
        unsigned short  dkg_intrlv;     /* interleave factor */
        unsigned short  dkg_gap1;       /* gap 1 size */
        unsigned short  dkg_gap2;       /* gap 2 size */
        unsigned short  dkg_apc;        /* alternates per cyl (SCSI only) */
        unsigned short  dkg_extra[9];   /* for compatible expansion */
};

/* disk io control commands */
#define  DKIOCGGEOM _IOR(d, 2, struct dk_geom)      /* Get geometry */
#define  DKIOCSGEOM         _IOW(d, 3, struct dk_geom)      /* Set geometry */
#define  DKIOCGPART         _IOR(d, 4, struct dk_map) /* Get partition info */
#define  DKIOCSPART         _IOW(d, 5, struct dk_map)       /* Set partition info */
#define  DKIOCINFO          _IOR(d, 8, struct dk_info) /* Get info */
```

The DKIOCGINFO ioctl returns a dk_info structure which tells the kind of the controller and attributes about how bad-block processing is done on the controller. The DKIOCGPART and DKIOCSPART get and set the controller's current notion of the partition table for the disk (without changing the partition table on the disk itself), while the DKIOCGGEOM and DKIOCSGEOM ioctl's do similar things for the per-drive geometry information.

SEE ALSO

ip(4S), sd(4S), xy(4S)

NAME

drum – paging device

SYNOPSIS

None; included with standard system.

DESCRIPTION

This file refers to the paging device in use by the system. This may actually be a subdevice of one of the disk drivers, but in a system with paging interleaved across multiple disk drives it provides an indirect driver for the multiple drives.

FILES

/dev/drum

BUGS

Reads from the drum are not allowed across the interleaving boundaries. Since these only occur every .5Mbytes or so, and since the system never allocates blocks across the boundary, this is usually not a problem.

NAME

ec – 3Com 10 Mb/s Ethernet interface

SYNOPSIS — SUN-2

device ec0 at mbmem ? csr 0xe0000 priority 3
device ec1 at mbmem ? csr 0xe2000 priority 3

DESCRIPTION

The *ec* interface provides access to a 10 Mb/s Ethernet network through a 3COM controller. For a general description of network interfaces see *if*(4N).

The hardware consumes 8 kilobytes of Multibus memory space. This memory is used for internal buffering by the board. The board starts at standard addresses 0xE0000 or 0xE2000. The board must be configured for interrupt level 3.

The interface software implements an exponential backoff algorithm when notified of a collision on the cable.

The interface handles the Internet protocol family, with the interface address maintained in Internet format. The Address Resolution Protocol *arp*(4P) is used to map 32-bit Internet addresses used in *inet*(4F) to the 48-bit addresses used on the Ethernet.

DIAGNOSTICS

ec%d: Ethernet jammed. After 16 failed transmissions and backoffs using the exponential backoff algorithm, the packet was dropped.

ec%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

arp(4P), if(4N), inet(4F)

BUGS

The interface hardware is not capable of talking to itself, making diagnosis more difficult.

NAME

fb – driver for Sun console frame buffer

SYNOPSIS

None; included in standard system.

DESCRIPTION

The *fb* driver provides indirect access to a Sun graphics controller board. It is an indirect driver for the Sun workstation console's frame buffer. At boot time, the workstation's frame buffer device is determined from information from the Monitor Proms and set to be the one that *fb* will indirect to. The device driver for the console's frame buffer must be configured into the kernel so that this indirect driver can access it.

The idea behind this driver is that user programs can open a known device, query its characteristics and access it in a device dependent way, depending on the type. *Fb* redirects *open*(2V), *close*(2), *ioctl*(2), and *mmap*(2) calls to the real frame buffer. All of the Sun frame buffers support the same general interface; see *fbio*(4S)

FILES

/dev/fb

SEE ALSO

fbio(4S), bwone(4S), bwtwo(4S), cgone(4S), cgtwo(4S), gpone(4S)

NAME

fbio – general properties of frame buffers

DESCRIPTION

All of the Sun frame buffers support the same general interface. Each responds to a FBIOGTYPE ioctl which returns information in a structure defined in *<sun/fbio.h>*:

```
struct   fbtype {
        int     fb_type;        /* as defined below */
        int     fb_height;      /* in pixels */
        int     fb_width;       /* in pixels */
        int     fb_depth;       /* bits per pixel */
        int     fb_cmsize;      /* size of color map (entries) */
        int     fb_size;        /* total size in bytes */
};

#define FBTYPE_SUN1BW       0
#define FBTYPE_SUN1COLOR    1
#define FBTYPE_SUN2BW       2
#define FBTYPE_SUN2COLOR    3
#define FBTYPE_SUN2GP       4
```

Each device has an FBTYPE which is used by higher-level software to determine how to perform raster-op and other functions. Each device is used by opening it, doing a FBIOGTYPE ioctl to see which frame buffer type is present, and thereby selecting the appropriate device-management routines.

Full-fledged frame buffers (that is, those that run SunWindows) implement an FBIOGPIXRECT ioctl, which returns a pixrect. This call is made only from inside the kernel. The returned pixrect is used by *win*(4S) for cursor tracking and colormap loading.

FBIOSVIDEO and FBIOGVIDEO are general-purpose ioctls for controlling possible video features of frame buffers. They are defined in *<sun/fbio.h>* . These ioctls either set or return the value of a flags integer. At this point, only the FBVIDEO_ON option is available, controlled by FBIOSVIDEO. FBIOGVIDEO returns this current video state.

The FBIOSATTR and FBIOGATTR ioctls allow access to special features of newer frame buffers. They use the following structures as defined in *<sun/fbio.h>*:

```
#define  FB_ATTR_NDEVSPECIFIC       8       /* no. of device specific values */
#define  FB_ATTR_NEMUTYPES          4       /* no. of emulation types */

struct fbsattr {
        int     flags;                          /* misc flags */
#define  FB_ATTR_AUTOINIT           1       /* emulation auto init flag */
#define  FB_ATTR_DEVSPECIFIC        2       /* dev. specific stuff valid flag */
        int     emu_type;                       /* emulation type (-1 if unused) */
        int     dev_specific[FB_ATTR_NDEVSPECIFIC];/* catchall */
};

struct fbgattr {
        int     real_type;                      /* real device type */
        int     owner;                          /* PID of owner, 0 if myself */
        struct fbtype fbtype;                   /* fbtype info for real device */
        struct fbsattr sattr;                   /* see above */
        int     emu_types[FB_ATTR_NEMUTYPES];   /* possible emulations */
                                                /* (-1 if unused) */
};
```

SEE ALSO

mmap(2), bwone(4S), bwtwo(4S), cgone(4S), cgtwo(4S), cgfour(4S), gpone(4S), fb(4S), win(4S)

BUGS

FBIOSATTR and FBIOGATTR are only supported by the *cgfour*(4S) frame buffer.

The FBVIDEO_ON flag my be incorrect for the *bwone*(4S), *bwtwo*(4S), and *cgfour*(4S) frame buffers since the drivers for these devices do not test the hardware, but simply report the last state stored by FBIOSVIDEO. The stored state and actual hardware state can get out of sync for several reasons: (1) processes having the same *bwone* frame buffer mapped can directly enable or disable the video, unknown to the driver; (2) */dev/bwtwo0* and */dev/cgfour0* on a *cgfour* CPU refer to the same frame buffer hardware; the video state of */dev/bwtwo0* may change, unknown to the *cgfour* driver; (3) if the hardware is not the default frame buffer */dev/fb*, the hardware's initial state is "video off", unknown to the driver.

NAME

gpone – Sun-3/Sun-2 graphics processor

SYNOPSIS — SUN-3

device gpone0 at vme24d16 ? csr

SYNOPSIS — SUN-2

device gpone0 at vme24 ? csr

DESCRIPTION

The *gpone* interface provides access to the optional Graphics Processor Board (GP).

The hardware consumes 64 kilobytes of VME bus address space. The GP board starts at standard address 0x210000 and must be configured for interrupt level 3.

GP IOCTL

The graphics processor responds to a number of ioctl calls as described here. One of the calls uses a **gp1fbinfo** structure that looks like this:

```
struct   gp1fbinfo {
             int          fb_vmeaddr;    /* physical color board address */
             int          fb_hwwidth;    /* fb board width */
             int          fb_hwheight;   /* fb board height */
             int          addrdelta;     /* phys addr diff between fb and gp */
             caddr_t      fb_ropaddr;    /* cg2 va thru kernelmap */
             int          fbunit;        /* fb unit to use  for a,b,c,d */
};
```

The ioctl call looks like this:

ioctl(file, request, argp)
int file, request;

argp is defined differently for each GP ioctl request and is specified in the descriptions below.

The following ioctl commands provide for transferring data between the graphics processor and color boards and processes.

GP1IO_PUT_INFO

Passes information about the frame buffer into driver. **argp** points to a **struct gp1fbinfo** which is passed to the driver.

GP1IO_GET_STATIC_BLOCK

Hands out a static block from the GP. **argp** points to an **int** which is returned from the driver.

GP1IO_FREE_STATIC_BLOCK

Frees a static block from the GP. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_GBUFFER_STATE

Checks to see if there is a buffer present on the GP. **argp** points to an **int** which is returned from the driver.

GP1IO_CHK_GP

Restarts the GP if necessary. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_RESTART_COUNT

Returns the number of restarts of a GP since power on. Needed to differentiate SIGXCPU calls in user processes. **argp** points to an **int** which is returned from the driver.

GP1IO_REDIRECT_DEVFB

Configures */dev/fb* to talk to a graphics processor device. **argp** points to an **int** which is passed to the driver.

GP1IO_GET_REQDEV

Returns the requested minor device. **argp** points to a **dev_t** which is returned from the driver.

GP1IO_GET_TRUMINORDEV
Returns the true minor device. **argp** points to a **char** which is returned from the driver.

The graphics processor driver also responds to the FBIOGTYPE, ioctl which a program can use to inquire as to the characteristics of the display device, the FBIOGINFO, ioctl for passing generic information, and the FBIOGPIXRECT ioctl so that SunWindows can run on it. See *fbio*(4S).

FILES

/dev/gpone[0-3][abcd]
/usr/include/sun/gpio.h
/usr/include/pixrect/{gp1cmds.h,gp1reg.h,gp1var.h}

SEE ALSO

fbio(4S), mmap(2), gpconfig(8)
Hardware Reference Manual for the Sun Graphics Processor (Sun 800-1190-01)
Software Interface Manual for the Sun Graphics Processor (Sun 800-1571-01)

DIAGNOSTICS

The Graphics Processor has been restarted. You may see display garbage as a result.

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

None; included automatically with *inet*(4F).

DESCRIPTION

The Internet Control Message Protocol, ICMP, is used by gateways and destination hosts which process datagrams to communicate errors in datagram-processing to source hosts. The datagram level of Internet is discussed in *ip*(4P). ICMP uses the basic support of IP as if it were a higher level protocol; however, ICMP is actually an integral part of IP. ICMP messages are sent in several situations; for example: when a datagram cannot reach its destination, when the gateway does not have the buffering capacity to forward a datagram, and when the gateway can direct the host to send traffic on a shorter route.

The Internet protocol is not designed to be absolutely reliable. The purpose of these control messages is to provide feedback about problems in the communication environment, not to make IP reliable. There are still no guarantees that a datagram will be delivered or that a control message will be returned. Some datagrams may still be undelivered without any report of their loss. The higher level protocols which use IP must implement their own reliability mechanisms if reliable communication is required.

The ICMP messages typically report errors in the processing of datagrams; for fragmented datagrams, ICMP messages are sent only about errors in handling fragment 0 of the datagram. To avoid the infinite regress of messages about messages etc., no ICMP messages are sent about ICMP messages. ICMP may however be sent in response to ICMP messages (for example, ECHOREPLY). There are eleven types of ICMP packets which can be received by the system. They are defined in this excerpt from <netinet/ip_icmp.h>, which also defines the values of some additional codes specifying the cause of certain errors.

```
/*
 * Definition of type and code field values
 */
#define ICMP_ECHOREPLY          0    /* echo reply */
#define ICMP_UNREACH            3    /* dest unreachable, codes: */
#define  ICMP_UNREACH_NET       0    /* bad net */
#define  ICMP_UNREACH_HOST      1    /* bad host */
#define  ICMP_UNREACH_PROTOCOL  2    /* bad protocol */
#define  ICMP_UNREACH_PORT      3    /* bad port */
#define  ICMP_UNREACH_NEEDFRAG  4    /* IP_DF caused drop */
#define  ICMP_UNREACH_SRCFAIL   5    /* src route failed */
#define ICMP_SOURCEQUENCH       4    /* packet lost, slow down */
#define ICMP_REDIRECT           5    /* shorter route, codes: */
#define  ICMP_REDIRECT_NET      0    /* for network */
#define  ICMP_REDIRECT_HOST     1    /* for host */
#define  ICMP_REDIRECT_TOSNET   2    /* for tos and net */
#define  ICMP_REDIRECT_TOSHOST  3    /* for tos and host */
#define ICMP_ECHO               8    /* echo service */
#define ICMP_TIMXCEED           11   /* time exceeded, code: */
#define  ICMP_TIMXCEED_INTRANS  0    /* ttl==0 in transit */
#define  ICMP_TIMXCEED_REASS    1    /* ttl==0 in reass */
#define ICMP_PARAMPROB          12   /* ip header bad */
#define ICMP_TSTAMP             13   /* timestamp request */
#define ICMP_TSTAMPREPLY        14   /* timestamp reply */
#define ICMP_IREQ               15   /* information request */
#define ICMP_IREQREPLY          16   /* information reply */
```

Arriving ECHO and TSTAMP packets cause the system to generate ECHOREPLY and TSTAMPREPLY packets. IREQ packets are not yet processed by the system, and are discarded. UNREACH, SOURCEQUENCH, TIMXCEED and PARAMPROB packets are processed internally by the protocols implemented in the system, or reflected to the user if a raw socket is being used; see *ip*(4P). REDIRECT, ECHOREPLY, TSTAMPREPLY and IREQREPLY are also reflected to users of raw sockets. In addition, REDIRECT messages cause the kernel routing tables to be updated; see *routing*(4N).

SEE ALSO

inet(4F), ip(4P)

Internet Control Message Protocol, RFC792, J. Postel, USC-ISI (Sun 800-1064-01)

BUGS

IREQ messages are not processed properly: the address fields are not set.

Messages which are source routed are not sent back using inverted source routes, but rather go back through the normal routing mechanisms.

NAME

ie – Intel 10 Mb/s Ethernet interface

SYNOPSIS — SUN-3

device ie0 at obio ? csr 0xc0000 priority 3
device ie1 at vme24d16 ? csr 0x88000 priority 3 vector ieintr 0x75

SYNOPSIS — SUN-2

device ie0 at obio 2 csr 0x7f0800 priority 3
device ie0 at mbmem ? csr 0x88000 priority 3
device ie1 at mbmem ? csr 0x8c000 flags 2 priority 3
device ie1 at vme24 ? csr 0x88000 priority 3 vector ieintr 0x75

DESCRIPTION

The *ie* interface provides access to a 10 Mb/s Ethernet network through the Intel 82586 controller chip. For a general description of network interfaces see *if*(4N).

In the synopsis—Sun-3 lines above, the first line specifies the first, CPU board-resident, Intel Ethernet interface; the second line specifies a second Intel interface present on a Sun Ethernet board.

In the synopsis—Sun-2 lines above, the first line specifies the first Intel Ethernet controller on a Sun-2/50 or Sun-2/160; the second line specifies the first Intel Ethernet controller on a Sun-2/120 or Sun-2/170.

DIAGNOSTICS

There are too many driver messages to list them all individually here. Some of the more common messages and their meanings follow.

ie%d: Ethernet jammed
: Network activity has become so intense that sixteen successive transmission attempts failed, causing the 82586 to give up on the current packet. Another possible cause of this message is a noise source somewhere in the network, such as a loose transceiver connection.

ie%d: no carrier
: The 82586 has lost input to its carrier detect pin while trying to transmit a packet, causing the packet to be dropped. Possible causes include an open circuit somewhere in the network and noise on the carrier detect line from the transceiver.

ie%d: lost interrupt: resetting
: The driver and 82586 chip have lost synchronization with each other. The driver recovers by resetting itself and the chip.

ie%d: iebark reset
: The 82586 failed to complete a watchdog timeout command in the alloted time. The driver recovers by resetting itself and the chip.

ie%d: WARNING: requeueing
: The driver has run out of resources while getting a packet ready to transmit. The packet is put back on the output queue for retransmission after more resources become available.

ie%d: panic: scb overwritten
: The driver has discovered that memory that should remain unchanged after initialization has become corrupted. This error usually is a symptom of a bad 82586 chip.

NAME

if – general properties of network interfaces

DESCRIPTION

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, *lo*(4), do not.

At boot time each interface which has underlying hardware support makes itself known to the system during the autoconfiguration process. Once the interface has acquired its address it is expected to install a routing table entry so that messages may be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR ioctl before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the ioctl; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (for example, 10Mb/s Ethernets using *arp*(4P),), the entire address specified in the ioctl is used.

The following *ioctl* calls may be used to manipulate network interfaces. Unless specified otherwise, the request takes an *ifreq* structure as its parameter. This structure has the form

```
struct    ifreq {
          char      ifr_name[16];           /* name of interface (e.g. "ec0") */
          union {
                    struct    sockaddr ifru_addr;
                    struct    sockaddr ifru_dstaddr;
                    short     ifru_flags;
          } ifr_ifru;
#define   ifr_addr          ifr_ifru.ifru_addr /* address */
#define   ifr_dstaddr       ifr_ifru.ifru_dstaddr     /* other end of p-to-p link */
#define   ifr_flags         ifr_ifru.ifru_flags /* flags */
};
```

SIOCSIFADDR
: Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address.

SIOCSIFDSTADDR
: Set point to point address for interface.

SIOCGIFDSTADDR
: Get point to point address for interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS
: Get interface flags.

SIOCGIFCONF
: Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
```

```
 */
struct  ifconf {
        int     ifc_len;            /* size of associated buffer */
        union {
                caddr_t  ifcu_buf;
                struct   ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf  ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req  ifc_ifcu.ifcu_req /* array of structures returned */
};
```

SEE ALSO

arp(4P), ec(4S), lo(4)

NAME

inet – Internet protocol family

SYNOPSIS

options INET

DESCRIPTION

The Internet protocol family is a collection of protocols layered atop the *Internet Protocol* (IP) transport layer, and using the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format (on the VAX these are word and byte reversed; on the Sun they are not reversed). The include file *<netinet/in.h>* defines the Internet address as a discriminated union.

Sockets in the Internet protocol family use the following addressing structure:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

(Library routines to return and manipulate structures of this form are in section 3N of the manual; see *intro*(3N) and the other section 3 entries mentioned under SEE ALSO below). Each socket has a local address which may be specified in this form, which can be established with *bind*(2); the *getsockname*(2) call returns this address. Each socket also may be bound to a peer socket with an address specified in this form; this peer address can be specified in a *connect*(2) call, or transiently with a single message in a *sendto* or *sendmsg* call; see *send*(2). The peer address of a socket is returned by the *getpeername*(2) call.

The sin_addr field of the socket address specifies the Internet address of the machine on which the socket is located. A special value may be specified or returned for this field, sin_addr.s_addr==INADDR_ANY. This address is a "wildcard" and matches any of the legal internet addresses on the local machine. This address is useful when a process neither knows (nor cares) what the local Internet address is, and even more useful for server processes which wish to service all requests of the current machine. Since a machine can have several addresses (one per hardware network interface), specifying a single address would restrict access to the service to those clients which specified the address of that interface. By specifying INADDR_ANY, the server can arrange to service clients from all interfaces.

When a socket address is bound, the networking system checks for an interface with the address specified on the current machine (unless, of course, a wildcard address is specified), and returns an error EADDRNOTAVAIL if no such interface is found.

The local port address specified in a *bind*(2) call is restricted to be greater than IPPORT_RESERVED (=1024, in <netinet/in.h>) unless the creating process is running as the super-user, providing a space of protected port numbers. The local port address is also required to not be in use in order for it to be assigned. This is checked by looking for another socket of the same type which has the same local address and local port number. If such a socket already exists, you will not be able to create another socket at the same address, and will instead get the error EADDRINUSE. If the local port address is specified as 0, then the system picks a unique port address not less than IPPORT_RESERVED and assigns it to the port. A unique local port address is also picked for a socket which is not bound but which is used with *connect*(2) or *sendto*(2); this allows *tcp*(4P) connections to be made by simply doing *socket*(2) and then *connect*(2) in the case where the local port address is not significant; it is defaulted by the system. Similarly if you are sending datagrams with *udp*(4P) and do not care which port they come from, you can just do *socket*(2) and *sendto*(2) and let the system pick a port number.

Let us say that two sockets are incompatible if they have the same port number, are not conected to other sockets, and do not have different local host addresses. (It is possible to have two sockets with the same port number and different local host addresses because a machine may have several local addresses from its different network interfaces.) The Internet system does not allow such incompatible sockets to exist on a single machine. Consider a socket which has a specific local host and local port number on the current machine. If another process tries to create a socket with a wildcard local host address and the same port number then that request will be denied. For connection based sockets this prevents these two sockets from attempting to connect to the same foreign host/socket, and thereby causing great havoc. For connectionless sockets this prevents the dilemma which would result from trying to determine who to deliver an incoming datagram to (since more than one socket could match an address given on a datagram). The same restriction applies if the wildcard socket exists first. (If both sockets are wildcard, then the normal restrictions on duplicate addresses apply.)

A socket option SO_REUSEADDR exists to allow incompatible sockets to be created. This option is needed to implement the File Transfer Protocol (FTP) which requires that a connection be made from an existing port number (the port number of its primary connection) to a different port number on the same remote host. The danger here is that the user would attempt to connect this second port to the same remote host/port that the primary connection was using. In using SO_REUSEADDR the user is pledging not to do this, since this will cause the first connection to abort.

When a *connect*(2) is done, the Internet system first checks that the socket is not already connected. If does not allow connections to port number 0 on another host, nor does it allow connections to a wildcard host (sin_addr.s_addr==INADDR_ANY); attempts to do this yield EADDRINUSE. If the socket from which the connection is being made currently has a wildcard local address (either because it was bound to a specific port with a wildcard address, or was never subjected to *bind*(2)), then the system picks a local Internet address for the socket from the set of addresses of interfaces on the local machine. If there is an interface on the local machine on the same network as the machine being connected to, then that address is used. Otherwise, the "first" local network interface is used (this is the one that prints out first in "netstat –i"; see *netstat*(8)). Although it is not supposed to matter which interface address is used, in practice it would probably be better to select the address of the interface through which the packets are to be routed. This is not currently done (as it would involve a fair amount of additional overhead for datagram transmission).

PROTOCOLS

The Internet protocol family supported by the operating system is comprised of the Internet Datagram Protocol (IP) *ip*(4P), Address Resolution Protocol (ARP) *arp*(4P), Internet Control Message Protocol (ICMP) *icmp*(4P), Transmission Control Protocol (TCP) *tcp*(4P), and User Datagram Protocol (UDP) *udp*(4P).

TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW; see *ip*(4P). The ICMP message protocol is most often used by the kernel to handle and report errors in protocol processing; it is, however, accessible to user programs. The ARP protocol is used to translate 32-bit Internet host numbers into the 48-bit addresses needed for an Ethernet.

SEE ALSO

intro(3N), byteorder(3N), gethostent(3N), getnetent(3N), getprotoent(3N), getservent(3N), inet(3N), arp(4P), tcp(4P), udp(4P), ip(4P)
Internet Protocol Transition Workbook, Network Information Center, SRI (Sun 800-1056-01)
Internet Protocol Implementation Guide, Network Information Center, SRI (Sun 800-1055-01)
A 4.2BSD Interprocess Communication Primer

NAME

ip – Internet Protocol

SYNOPSIS

None; included by default with *inet*(4F).

DESCRIPTION

The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. It provides for transmitting blocks of data called "datagrams" from sources to destinations, where sources and destinations are hosts identified by fixed-length addresses. It also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.

IP is specifically limited in scope. There are no mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. IP can capitalize on the services of its supporting networks to provide various types and qualities of service.

IP is called on by host-to-host protocols, including *tcp*(4P) a reliable stream protocol, *udp*(4P) a socket-socket datagram protocol, and *nd*(4P) the network disk protocol. Other protocols may be layered on top of IP using the *raw* protocol facilities described here to receive and send datagrams with a specific IP protocol number. The IP protocol calls on local network drivers to carry the internet datagram to the next gateway or destination host.

When a datagram arrives at a UNIX system host, the system performs a checksum on the header of the datagram. If this fails, or if the datagram is unreasonably short or the header length specified in the datagram is not within range, then the datagram is dropped. Checksumming of Internet datagrams may be disabled for debugging purposes by patching the kernel variable *ipcksum* to have the value 0.

Next the system scans the IP options of the datagram. Options allowing for source routing (see *routing*(4N)) and also the collection of time stamps as a packet follows a particular route (for network monitoring and statistics gathering purposes) are handled; other options are ignored. Processing of source routing options may result in an UNREACH *icmp*(4P) message because the source routed host is not accessible.

After processing the options, IP checks to see if the current machine is the destination for the datagram. If not, then IP attempts to forward the datagram to the proper host. Before forwarding the datagram, IP decrements the time to live field of the datagram by IPTTLDEC seconds (currently 5 from <netinet/ip.h>), and discards the datagram if its lifetime has expired, sending an ICMP TIMXCEED error packet back to the source host. Similarly if the attempt to forward the datagram fails, then ICMP messages indicating an unreachable network, datagram too large, unreachable port (datagram would have required broadcasting on the target interface, and IP does not allow directed broadcasts), lack of buffer space (reflected as a source quench), or unreachable host. Note however, in accordance with the ICMP protocol specification, ICMP messages are returned only for the first fragment of fragmented datagrams.

It is possible to disable the forwarding of datagrams by a host by patching the kernel variable *ipforwarding* to have value 0.

If a packet arrives and is destined for this machine, then IP must check to see if other fragments of the same datagram are being held. If this datagram is complete, then any previous fragments of it are discarded. If this is only a fragment of a datagram, it may yield a complete set of pieces for the datagram, in which case IP constructs the complete datagram and continues processing with that. If there is yet no complete set of pieces for this datagram, then all data thus far received is held (but only one copy of each data byte from the datagram) in hopes that the rest of the pieces of the fragmented datagram will arrive and we will be able to proceed. We allow IPFRAGTTL (currently 15 in <netinet/ip.h>) seconds for all the fragments of a datagram to arrive, and discard partial fragments then if the datagram has not yet been completely assembled.

When we have a complete input datagram it is passed out to the appropriate protocol's input routine: either *tcp*(4P), *udp*(4P), *nd*(4P), *icmp*(4P) or a user process through a raw IP socket as described below.

Datagrams are output by the system-implemented protocols *tcp*(4P), *udp*(4P), *nd*(4P), and *icmp*(4P); as well as by packet forwarding operations and user processes through raw IP sockets. Output packets are normally subjected to routing as described in *routing*(4N). However, special processes such as the routing daemon *routed*(8C) occasionally use the SO_DONTROUTE socket option to make packets avoid the routing tables and go directly to the network interface with the network number which the packet is addressed to. This may be used to test the ability of the hardware to transmit and receive packets even when we believe that the hardware is broken and have therefore deleted it from the routing tables.

If there is no route to a destination address or if the SO_DONTROUTE option is given and there is no interface on the network specified by the destination address, then the IP output routine returns a ENETUNREACH error. (This and the other IP output errors are reflected back to user processes through the various protocols, which individually describe how errors are reported.)

In the (hopefully normal) case where there is a suitable route or network interface, the destination address is checked to see if it specifies a broadcast (address INADDR_ANY; see *inet*(4F)); if it does, and the hardware interface does not support broadcasts, then an EADDRNOTAVAIL is returned; if the caller is not the super-user then a EACCESS error will be returned. IP also does not allow broadcast messages to be fragmented, returning a EMSGSIZE error in this case.

If the datagram passes all these tests, and is small enough to be sent in one chunk, then the system calls the output routine for the particular hardware interface to transmit the packet. The interface may give an error indication, which is reflected to IP output's caller; see the documentation for the specific interface for a description of errors it may encounter. If a datagram is to be fragmented, it may have the IP_DF (don't fragment) flag set (although currently this can happen only for forwarded datagrams). If it does, then the datagram will be rejected (and result in an ICMP error datagram). If the system runs out of buffer space in fragmenting a datagram then a ENOBUFS error will be returned.

IP provides a space of 255 protocols. The known protocols are defined in <netinet/in.h>. The ICMP, TCP, UDP and ND protocols are processed internally by the system; others may be accessed through a raw socket by doing:

s = socket(AF_INET, SOCK_RAW, IPPROTO_xxx);

Datagrams sent from this socket will have the current host's address and the specified protocol number; the raw IP driver will construct an appropriate header. When IP datagrams are received for this protocol they are queued on the raw socket where they may be read with *recvfrom;* the source IP address is reflected in the received address.

SEE ALSO

send(2), recv(2), inet(4F)

Internet Protocol, RFC791, USC-ISI (Sun 800-1063-01)

BUGS

One should be able to send and receive IP options.

Raw sockets should receive ICMP error packets relating to the protocol; currently such packets are simply discarded.

NAME

ip – Disk driver for Interphase 2180 SMD Disk Controller

SYNOPSIS — SUN-2

controller ipc0 at mbio ? csr 0x040 priority 2
controller ipc1 at mbio ? csr 0x044 priority 2
disk ip0 at ipc0 drive0
disk ip1 at ipc0 drive1
disk ip2 at ipc1 drive0
disk ip3 at ipc1 drive1

DESCRIPTION

Special files *ip** refer to disk devices controlled by an Interphase SMD 2180 disk controller.

The standard *ip* device names begin with the letters "ip", followed by the drive unit number, followed by a letter from the series a – h to name one of the eight partitions on the drive. For example, */dev/ip1c* refers to partition c on the second drive controlled by the Interphase controller.

The device names provide the binding into the minor device numbers for the driver software. Files with minor device numbers 0 through 7 refer to the eight partitions (a – h) of unit 0; files with device numbers 8 through 15 refer to the eight partitions of drive 1, and so on.

The block files access the disk via the system's normal buffering mechanism, and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more effficient when many words are transmitted. Raw files conventionally have a leading "r" — */dev/rip0c*, for instance.

In raw I/O, counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

DISK SUPPORT

This driver handles all SMD drives by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The ip?a partition is normally used for the root file system on a disk, the ip?b partition as a paging area, and the ip?c partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the ip?g partition.

FILES

/dev/ip[0-7][a-h] block files
/dev/rip[0-7][a-h] raw files

SEE ALSO

dkio(4S), xy(4S)

DIAGNOSTICS

ip*n*: SMD-2180
: When booting tells the controller type.

ip*n* : initialization failed
: Because the controller didn't respond; perhaps another device is at the address the system expected an Interphase controller at.

ip*n* : error *n* reading label on head *n*
: Error reading drive geometry/partition table information.

ip*n* : Corrupt label on head *n*
: The geometry/partition label checksum was incorrect.

ip*n* : Misplaced label on head *n*
: A disk label was copied to the wrong head on the disk; shoudn't happen.

ip*n*: Unsupported phys partition # *n*
This indicates a bad label.

ip*n*: unit not online

ip*n c*: *cmd how (msg)* blk *n*
A command such as read, write, or format encountered a error condition (how): either it *failed*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready", "sector not found" or "disk write protected".

BUGS

In raw I/O *read* and *write*(2V) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

NAME

kb, kbd – Sun keyboard

SYNOPSIS

pseudo-device kb*number*

DESCRIPTION

kb provides access to the Sun workstation keyboard and its translation tables. Definitions for altering keyboard translation, and reading events from the keyboard, are in **<sundev/kbio.h>** and **<sundev/kbd.h>**. *number* specifies the maximum number of keyboards supported by the system. In addition, the kernel recognizes the special keyboard device **/dev/kbd**; it is a synonym for the system keyboard, which is physically attached to the system console. This keyboard's keystrokes are normally treated as input to the **/dev/console** device.

The UNIX kernel recognizes which keys have been typed using a set of tables for each known type of keyboard. Each translation table is an array of 128 bytes (unsigned characters). If a character value is less than 0x80, it is treated as an ASCII character (perhaps with the META bit included). Higher values indicate special characters that invoke more complicated actions.

Keyboard Translation State

The call KIOCTRANS controls the presence of keyboard translation, for which the following values are defined:

```
#define TR_NONE             0
#define TR_ASCII            1
#define TR_EVENT            2
#define TR_UNTRANS_EVENT    3

int x;
err = ioctl(fd, KIOCTRANS, &x);
```

When *x* is TR_NONE, keyboard translation is turned off and up/down key codes are reported. Specifying *x* as TR_ASCII causes ASCII to be reported. Specifying *x* as TR_EVENT causes *Firm_events* to be reported (see below). Specifying *x* as TR_UNTRANS_EVENT gives unencoded keyboard values for all input events within the window system.

Keyboard Translation-Table Entries

The call KIOCSETKEY changes a keyboard translation table entry, using the *kiockey* struct:

```
struct  kiockey {
        int     kio_tablemask;  /* Translation table (one of: 0, CAPSMASK,
                                   SHIFTMASK, CTRLMASK, UPMASK) */
#define KIOCABORT1   -1         /* Special "mask": abort1 keystation */
#define KIOCABORT2   -2         /* Special "mask": abort2 keystation */
        u_char  kio_station;    /* Physical keyboard key station (0-127) */
        u_char  kio_entry;      /* Translation table station's entry */
        char    kio_string[10]; /* Value for STRING entries (null terminated) */
};

struct kiockey key;
err = ioctl(fd, KIOCSETKEY, &key);
```

To alter a keyboard translation-table entry, set *kio_tablemask* table's *kio_station* to *kio_entry*. Copy *kio_string* to the string table if *kio_entry* is between STRING and STRING+15. This call may return EINVAL if there are invalid arguments.

There are a couple special values of *kio_tablemask* that affect the two step "break to the prom monitor" sequence. The usual sequence is **SETUP–a** or **L1–a**. If *kio_tablemask* is KIOCABORT1 then the value of *kio_station* is set to be the first keystation in the sequence. If *kio_tablemask* is KIOCABORT2 then the value of *kio_station* is set to be the second keystation in the sequence.

The call KIOCGETKEY determines the current value of a keyboard translation table entry:

```
struct kiockey key;
err = ioctl(fd, KIOCGETKEY, &key);
```

Get *kio_tablemask* table's *kio_station* to *kio_entry*. Get *kio_string* from string table if *kio_entry* is between STRING and STRING+15. This call may return EINVAL if there are invalid arguments.

Keyboard Type

The call KIOCTYPE returns the type of the keyboard:

```
#define  KB_KLUNK    0x00      /* Micro Switch 103SD32-2 */
#define  KB_VT100    0x01      /* Keytronics VT100 compatible */
#define  KB_SUN2     0x02      /* Sun-2 custom keyboard */
#define  KB_SUN3     0x03      /* Sun-3 custom keyboard */
#define  KB_ASCII    0x0F      /* ASCII terminal masquerading as kbd */

int x;
err = ioctl(fd, KIOCTYPE, &x);
```

When *x* is −1, the keyboard type is unknown.

Reading From The Keyboard

Normally, keystrokes are discarded, except for those typed at the system keyboard; those are translated and treated as input on the system console device **/dev/console**. In order to read keystrokes directly, the call KIOCSDIRECT must be used to set the keyboard to "direct input" mode. In this mode, keystrokes are translated and queued to be read by a process that has opened a keyboard device:

```
int on = 1;
err = ioctl(fd, KIOCSDIRECT, &on);
```

The KIOCSDIRECT call turns "direct input" mode on or off, depending on whether the variable pointed to by its argument has the value 1 or 0. The call KIOCGDIRECT sets the variable pointed to by its argument to the current state of this mode:

```
int direct_state;
err = ioctl(fd, KIOCGDIRECT, &direct_state);
```

When the keyboard device is closed, "direct mode" is turned off.

Keyboard Commands

The call KIOCCMD sends a command to the keyboard:

```
/*
 * Commands to the Sun-2 keyboard.
 */
#define KBD_CMD_RESET     0x01      /* Reset keyboard as if power-up */
#define KBD_CMD_BELL      0x02      /* Turn on the bell */
#define KBD_CMD_NOBELL    0x03      /* Turn off the bell */

/*
 * Commands to the Sun-3 keyboard.  KBD_CMD_BELL & KBD_CMD_NOBELL work
 * as well.
 */
#define KBD_CMD_CLICK     0x0A      /* Turn on the click annunciator */
#define KBD_CMD_NOCLICK   0x0B      /* Turn off the click annunciator */

int x;
err = ioctl(fd, KIOCCMD, &x);
```

Inappropriate commands for particular keyboard types are ignored. Since there is no reliable way to get the state of the bell or click (because we can't query the keyboard, and also because a process could do writes to the appropriate serial driver — thus going around this *ioctl*) we don't provide an equivalent *ioctl* to query its state.

Shift Masks

When shift keys are pressed or locked, a different translation table is used to translate keyboard actions. The shift mask indicates which translation table to use. Since there may be more than one bit in in the shift mask at any given time, they are prioritized as follows:

UPMASK 0x0080	"Key Up" translation table.
CTRLMASK 0x0030	"Controlled" translation table.
SHIFTMASK 0x000E	"Shifted" translation table.
CAPSMASK 0x0001	"Caps Lock" translation table.
(No shift keys pressed or locked)	"Unshifted" translation table.

That is: if the event corresponds to a key-up, use the "Key Up" table. If the CTRL is down, use the "Controlled" table, and so on.

Special-Entry Values

Special-entry values are classified according to the value of the high-order bits (with the exception of ALT, which is defined as 0x6). The high-order value for each class is defined as a constant, as shown in the list below. The value of the low-order bits, when added to this constant, distinguishes between keys within each class:

SHIFTKEYS 0x80	A shift key. The value of the particular shift key is added to determine which shift mask to apply:	
	CAPSLOCK 0	"Caps Lock" key.
	SHIFTLOCK 1	"Shift Lock" key.
	LEFTSHIFT 2	Left-hand "Shift" key.
	RIGHTSHIFT 3	Right-hand "Shift" key.
	LEFTCTRL 4	Left-hand (or only) "Control" key.
	RIGHTCTRL 5	Right-hand "Control" key.
BUCKYBITS 0x90	Used to toggle mode-key-up/down status without altering the value of an accompanying ASCII character. (The actual bit-position value, minus 7, is added.)	
	METABIT 0	The "Meta" key was pressed along with the key. This is the only user-accessible bucky bit.
	SYSTEMBIT 1	The "System" key was pressed. This is a place holder to indicate which key is the system-abort key.
FUNNY 0xA0	Performs various functions depending on the value of the low 4 bits:	
	NOP 0xA0	Does nothing.
	OOPS 0xA1	Exists, but is undefined.
	HOLE 0xA2	There is no key in this position on the keyboard, and the position-code should not be used.
	NOSCROLL 0xA3	Alternately sends ^S and ^Q.
	CTRLS 0xA4	Sends ^S and toggles NOSCROLL key.
	CTRLQ 0xA5	Sends ^Q and toggles NOSCROLL key.
	RESET 0xA6	Keyboard reset.

ERROR 0xA7	The keyboard driver detected an internal error.
IDLE 0xA8	The keyboard is idle (no keys down).
0xA9 — 0xAF	Reserved for nonparameterized functions.

STRING 0xB0 — 0xBF

The low-order bits index a table of strings. Each null-terminated string is returned character by character. The maximum length is defined as:

KTAB_STRLEN 10

Individual string numbers are defined as:
HOMEARROW 0x00
UPARROW 0x01
DOWNARROW 0x02
LEFTARROW 0x03
RIGHTARROW 0x04
String numbers 5 — F are available for custom entries.

Function Key Groups

In the following groups, the low-order bits indicate the function key number within the group:

LEFTFUNC	0xC0
RIGHTFUNC	0xD0
TOPFUNC	0xE0
BOTTOMFUNC	0xF0
LF(*n*)	(LEFTFUNC+(*n*)-1)
RF(*n*)	(RIGHTFUNC+(*n*)-1)
TF(*n*)	(TOPFUNC+(*n*)-1)
BF(*n*)	(BOTTOMFUNC+(*n*)-1)

There are 64 keys reserved for function keys. The actual positions may not be on left/right/top/bottom of the keyboard, although they usually are.

Normally, when a function key is pressed, the following escape sequence is sent:

<ESC>[0....9z

where <ESC> is a single escape character and "0...9" indicates the decimal representation of the function-key value.

INDEX STRUCTURES

There is a hierarchy of structures for accessing keyboard translation data. The array *keytables* contains pointers to the translation data for each of the known keyboard types:

```
struct keyboard *keytables[ ] = {
        &keyindex_ms,
        &keyindex_vt,
        &keyindex_s2,
        &keyindex_s3,
};
```

Each keyboard type is described by a **struct keyboard** that contains pointers to the five translation-tables ("Unshifted", "Shifted", "Caps Locked" "Controlled", and "Key Up") associated with that type, plus bit-masks that indicate what state can persist with no keys pressed, and the key-pair used as the abort sequence for the system.

An array *keystringtab* contains the strings sent by various keys, and can be accessed by any translation:

```
#define kstescinit(c)   {'\033', '[', 'c', '\0'}
char keystringtab[16][KTAB_STRLEN] = {
        kstescinit(H),                  /* home */
        kstescinit(A),                  /* up */
        kstescinit(B),                  /* down */
        kstescinit(D),                  /* left */
        kstescinit(C),                  /* right */
};
```

Index Structure for the Sun-3 Keyboard

```
static struct keyboard keyindex_s3 = {
        &keytab_s3_lc,
        &keytab_s3_uc,
        &keytab_s3_cl,
        &keytab_s3_ct,
        &keytab_s3_up,
        0x0000,                 /* Shift bits that stay on with idle keyboard */
        0x0000,                 /* Bucky bits that stay on with idle keyboard */
        0x01, 0x4d,             /* Abort sequence  L1-A */
        CAPSMASK,               /* Shift bits that toggle on down event */
};
```

Index Structure for the Sun-2 Keyboard

```
static struct keyboard keyindex_s2 = {
        &keytab_s2_lc,
        &keytab_s2_uc,
        &keytab_s2_cl,
        &keytab_s2_ct,
        &keytab_s2_up,
        CAPSMASK,               /* Shift bits that stay on with idle keyboard */
        0x0000,                 /* Bucky bits that stay on with idle keyboard */
        0x01, 0x4d,             /* Abort sequence L1-A */
        0x0000,                 /* Shift bits that toggle on down event */
};
```

Index Structure for the Micro Switch 103SD32-2 Keyboard

```
static struct keyboard keyindex_ms = {
        &keytab_ms_lc,
        &keytab_ms_uc,
        &keytab_ms_cl,
        &keytab_ms_ct,
        &keytab_ms_up,
        CTLSMASK,               /* Shift bits that stay on with idle keyboard */
        0x0000,                 /* Bucky bits that stay on with idle keyboard */
        0x01, 0x4d,             /* Abort sequence L1-A */
        0x0000,                 /* Shift bits that toggle on down event */
};
```

Index Structure for the VT100-Style Keyboard

```
static struct keyboard keyindex_vt = {
        &keytab_vt_lc,
        &keytab_vt_uc,
        &keytab_vt_cl,
        &keytab_vt_ct,
        &keytab_vt_up,
        CAPSMASK+CTLSMASK,   /* Shift keys that stay on with idle keyboard */
        0x0000,              /* Bucky bits that stay on with idle keyboard */
        0x01, 0x3b,          /* Abort sequence SETUP-A */
        0x0000,              /* Shift bits that toggle on down event */
};
```

DEFAULT TRANSLATION TABLES

Sun-3 Keyboard

Unshifted

Key Value	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*
00 HOLE	01 BUCKYBITS+ SYSTEMBIT	02 HOLE	03 LF(2)	04 HOLE	05 TF(1)	06 TF(2)	07 HOLE
08 TF(3)	09 HOLE	0A TF(4)	0B HOLE	0C TF(5)	0D HOLE	0E TF(6)	0F HOLE
10 TF(7)	11 TF(8)	12 TF(9)	13 ALT	14 HOLE	15 RF(1)	16 RF(2)	17 RF(3)
18 HOLE	19 LF(3)	1A LF(4)	1B HOLE	1C HOLE	1D c('[')	1E '1'	1F '2'
20 '3'	21 '4'	22 '5'	23 '6'	24 '7'	25 '8'	26 '9'	27 '0'
28 '–'	29 '='	2A '^'	2B '\b'	2C HOLE	2D RF(4)	2E RF(5)	2F RF(6)
30 HOLE	31 LF(5)	32 HOLE	33 LF(6)	34 HOLE	35 '\t'	36 'q'	37 'w'
38 'e'	39 'r'	3A 't'	3B 'y'	3C 'u'	3D 'i'	3E 'o'	3F 'p'
40 '['	41 ']'	42 0x7F	43 HOLE	45 RF(7)	45 STRING+ UPARROW	46 RF(9)	47 HOLE
48 LF(7)	49 LF(8)	4A LF(40)	4B HOLE	4C SHIFTKEYS+ LEFTCTRL	4D 'a'	4E 's'	4F 'd'
50 'f'	51 'g'	52 'h'	53 'j'	54 'k'	55 'l'	56 ';'	57 '\''
58 '\'	59 '\r'	5A HOLE	5B STRING+ LEFTARROW	5C RF(11)	5D STRING+ RIGHTARROW	5E HOLE	5F LF(9)
60 LF(15)	61 LF(10)	62 HOLE	63 SHIFTKEYS+ LEFTSHIFT	64 'z'	65 'x'	66 'c'	67 'v'
68 'b'	69 'n'	6A 'm'	6B ','	6C '.'	6D '/'	6E SHIFTKEYS+ RIGHTSHIFT	6F '\n'
70 RF(13)	71 STRING+ DOWNARROW	72 RF(15)	73 HOLE	74 HOLE	75 HOLE	76 HOLE	77 SHIFTKEYS+ CAPSLOCK
78 BUCKYBITS+ METABIT	79 ' '	7A BUCKYBITS+ METABIT	7B HOLE	7C HOLE	7D HOLE	7E ERROR	7F IDLE

Sun-3 Keyboard

Shifted

Key Value	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*
00 HOLE	01 BUCKYBITS+ SYSTEMBIT	02 HOLE	03 LF(2)	04 HOLE	05 TF(1)	06 TF(2)	07 HOLE
08 TF(3)	09 HOLE	0A TF(4)	0B HOLE	0C TF(5)	0D HOLE	0E TF(6)	0F HOLE
10 TF(7)	11 TF(8)	12 TF(9)	13 ALT	14 HOLE	15 RF(1)	16 RF(2)	17 RF(3)
18 HOLE	19 LF(3)	1A LF(4)	1B HOLE	1C HOLE	1D c('[')	1E '!'	1F '@'
20 '#'	21 '$'	22 '%'	23 '^'	24 '&'	25 '*'	26 '('	27 ')'
28 '_'	29 '+'	2A '~'	2B '\b'	2C HOLE	2D RF(4)	2E RF(5)	2F RF(6)
30 HOLE	31 LF(5)	32 HOLE	33 LF(6)	34 HOLE	35 '\t'	36 'Q'	37 'W'
38 'E'	39 'R'	3A 'T'	3B 'Y'	3C 'U'	3D 'I'	3E 'O'	3F 'P'
40 '{'	41 '}'	42 0x7F	43 HOLE	44 RF(7)	45 STRING+ UPARROW	46 RF(9)	47 HOLE
48 LF(7)	49 LF(8)	4A HOLE	4B HOLE	4C SHIFTKEYS+ LEFTCTRL	4D 'A'	4E 'S'	4F 'D'
50 'F'	51 'G'	52 'H'	53 'J'	54 'K'	55 'L'	56 ':'	57 '"'
58 '|'	59 '\r'	5A HOLE	5B STRING+ LEFTARROW	5C RF(11)	5D STRING+ RIGHTARROW	5E HOLE	5F LF(9)
60 LF(15)	61 LF(10)	62 HOLE	63 SHIFTKEYS+ LEFTSHIFT	64 'Z'	65 'X'	66 'C'	67 'V'
68 'B'	69 'N'	6A 'M'	6B '<'	6C '>'	6D '?'	6E SHIFTKEYS+ RIGHTSHIFT	6F '\n'
70 RF(13)	71 STRING+ DOWNARROW	72 RF(15)	73 HOLE	74 HOLE	75 HOLE	76 HOLE	77 SHIFTKEYS+ CAPSLOCK
78 BUCKYBITS+ METABIT	79 ' '	7A BUCKYBITS+ METABIT	7B HOLE	7C HOLE	7D HOLE	7E ERROR	7F IDLE

Caps Locked

Key Value	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*	*Key Value*
00 HOLE	01 BUCKYBITS+ SYSTEMBIT	02 HOLE	03 LF(2)	04 HOLE	05 TF(1)	06 TF(2)	07 HOLE
08 TF(3)	09 HOLE	0A TF(4)	0B HOLE	0C TF(5)	0D HOLE	0E TF(6)	0F HOLE
10 TF(7)	11 TF(8)	12 TF(9)	13 ALT	14 HOLE	15 RF(1)	16 RF(2)	17 RF(3)
18 HOLE	19 LF(3)	1A LF(4)	1B HOLE	1C HOLE	1D c('[')	1E '1'	1F '2'
20 '3'	21 '4'	22 '5'	23 '6'	24 '7'	25 '8'	26 '9'	27 '0'
28 '-'	29 '='	2A '`'	2B '\b'	2C HOLE	2D RF(4)	2E RF(5)	2F RF(6)
30 HOLE	31 LF(5)	32 HOLE	33 LF(6)	34 HOLE	35 '\t'	36 'Q'	37 'W'
38 'E'	39 'R'	3A 'T'	3B 'Y'	3C 'U'	3D 'I'	3E 'O'	3F 'P'
40 '['	41 ']'	42 0x7F	43 HOLE	44 RF(7)	45 STRING+ UPARROW	46 RF(9)	47 HOLE
48 LF(7)	49 LF(8)	4A HOLE	4B HOLE	4C SHIFTKEYS+ LEFTCTRL	4D 'A'	4E 'S'	4F 'D'
50 'F'	51 'G'	52 'H'	53 'J'	54 'K'	55 'L'	56 ';'	57 '\''
58 '\'	59 '\r'	5A HOLE	5B STRING+ LEFTARROW	5C RF(11)	5D STRING+ RIGHTARROW	5E HOLE	5F LF(9)
60 LF(15)	61 LF(10)	62 HOLE	63 SHIFTKEYS+ LEFTSHIFT	64 'Z'	65 'X'	66 'C'	67 'V'
68 'B'	69 'N'	6A 'M'	6B ','	6C '.'	6D '/'	6E SHIFTKEYS+ RIGHTSHIFT	6F '\n'
70 RF(13)	71 STRING+ DOWNARROW	72 RF(15)	73 HOLE	74 HOLE	75 HOLE	76 HOLE	77 SHIFTKEYS+ CAPSLOCK
78 BUCKYBITS+ METABIT	79 ' '	7A BUCKYBITS+ METABIT	7B HOLE	7C HOLE	7D HOLE	7E ERROR	7F IDLE

Sun-3 Keyboard

Controlled

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	LF(2)	04	HOLE	05	TF(1)	06	TF(2)	07	HOLE
08	TF(3)	09	HOLE	0A	TF(4)	0B	HOLE	0C	TF(5)	0D	HOLE	0E	TF(6)	0F	HOLE
10	TF(7)	11	TF(8)	12	TF(9)	13	ALT	14	HOLE	15	RF(1)	16	RF(2)	17	RF(3)
18	HOLE	19	LF(3)	1A	LF(4)	1B	HOLE	1C	HOLE	1D	c('[')	1E	'1'	1F	c('@')
20	'3'	21	'4'	22	'5'	23	c('^')	24	'7'	25	'8'	26	'9'	27	'0'
28	c('_')	29	'='	2A	c('~')	2B	'\b'	2C	HOLE	2D	RF(4)	2E	RF(5)	2F	RF(6)
30	HOLE	31	LF(5)	32	HOLE	33	LF(6)	34	HOLE	35	'\t'	36	c('q')	37	c('w')
38	c('e')	39	c('r')	3A	c('t')	3B	c('y')	3C	c('u')	3D	c('i')	3E	c('o')	3F	c('p')
40	c('[')	41	c(']')	42	0x7F	43	HOLE	44	RF(7)	45	STRING+ UPARROW	46	RF(9)	47	HOLE
48	LF(7)	49	LF(8)	4A	HOLE	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	c('a')	4E	c('s')	4F	c('d')
50	c('f')	51	c('g')	52	c('h')	53	c('j')	54	c('k')	55	c('l')	56	';'	57	'\''
58	c('\')	59	'\r'	5A	HOLE	5B	STRING+ LEFTARROW	5C	RF(11)	5D	STRING+ RIGHTARROW	5E	HOLE	5F	LF(9)
60	LF(15)	61	LF(10)	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	c('z')	65	c('x')	66	c('c')	67	c('v')
68	c('b')	69	c('n')	6A	c('m')	6B	','	6C	'.'	6D	c('_')	6E	SHIFTKEYS+ RIGHTSHIFT	6F	'\n'
70	RF(13)	71	STRING+ DOWNARROW	72	RF(15)	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	SHIFTKEYS+ CAPSLOCK
78	BUCKYBITS+ METABIT	79	c(' ')	7A	BUCKYBITS+ METABIT	7B	HOLE	7C	HOLE	7D	HOLE	7E	ERROR	7F	IDLE

Key Up

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	OOPS	04	HOLE	05	OOPS	06	OOPS	07	HOLE
08	OOPS	09	HOLE	0A	OOPS	0B	HOLE	0C	OOPS	0D	HOLE	0E	OOPS	0F	HOLE
10	OOPS	11	OOPS	12	OOPS	13	OOPS	14	HOLE	15	OOPS	16	OOPS	17	NOP
18	HOLE	19	OOPS	1A	OOPS	1B	HOLE	1C	HOLE	1D	NOP	1E	NOP	1F	NOP
20	NOP	21	NOP	22	NOP	23	NOP	24	NOP	25	NOP	26	NOP	27	NOP
28	NOP	29	NOP	2A	NOP	2B	NOP	2C	HOLE	2D	OOPS	2E	OOPS	2F	NOP
30	HOLE	31	OOPS	32	HOLE	33	OOPS	34	HOLE	35	NOP	36	NOP	37	NOP
38	NOP	39	NOP	3A	NOP	3B	NOP	3C	NOP	3D	NOP	3E	NOP	3F	NOP
40	NOP	41	NOP	42	NOP	43	HOLE	44	OOPS	45	OOPS	46	NOP	47	HOLE
48	OOPS	49	OOPS	4A	HOLE	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	NOP	4E	NOP	4F	NOP
50	NOP	51	NOP	52	NOP	53	NOP	54	NOP	55	NOP	56	NOP	57	NOP
58	NOP	59	NOP	5A	HOLE	5B	OOPS	5C	OOPS	5D	NOP	5E	HOLE	5F	OOPS
60	OOPS	61	OOPS	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	NOP	65	NOP	66	NOP	67	NOP
68	NOP	69	NOP	6A	NOP	6B	NOP	6C	NOP	6D	NOP	6E	SHIFTKEYS+ RIGHTSHIFT	6F	NOP
70	OOPS	71	OOPS	72	NOP	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	NOP
78	BUCKYBITS+ METABIT	79	NOP	7A	BUCKYBITS+ METABIT	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	RESET

Sun-2 Keyboard

Unshifted

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(11)	03	LF(2)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(11)
08	TF(3)	09	TF(12)	0A	TF(4)	0B	TF(13)	0C	TF(5)	0D	TF(14)	0E	TF(6)	0F	TF(15)
10	TF(7)	11	TF(8)	12	TF(9)	13	TF(10)	14	HOLE	15	RF(1)	16	RF(2)	17	RF(3)
18	HOLE	19	LF(3)	1A	LF(4)	1B	LF(12)	1C	HOLE	1D	c('[')	1E	'1'	1F	'2'
20	'3'	21	'4'	22	'5'	23	'6'	24	'7'	25	'8'	26	'9'	27	'0'
28	'-'	29	'='	2A	'`'	2B	'\b'	2C	HOLE	2D	RF(4)	2E	RF(5)	2F	RF(6)
30	HOLE	31	LF(5)	32	LF(13)	33	LF(6)	34	HOLE	35	'\t'	36	'q'	37	'w'
38	'e'	39	'r'	3A	't'	3B	'y'	3C	'u'	3D	'i'	3E	'o'	3F	'p'
40	'['	41	']'	42	0x7F	43	HOLE	44	RF(7)	45	STRING+ UPARROW	46	RF(9)	47	HOLE
48	LF(7)	49	LF(8)	4A	LF(14)	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	'a'	4E	's'	4F	'd'
50	'f'	51	'g'	52	'h'	53	'j'	54	'k'	55	'l'	56	';'	57	'\''
58	'\'	59	'\r'	5A	HOLE	5B	STRING+ LEFTARROW	5C	RF(11)	5D	STRING+ RIGHTARROW	5E	HOLE	5F	LF(9)
60	LF(15)	61	LF(10)	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	'z'	65	'x'	66	'c'	67	'v'
68	'b'	69	'n'	6A	'm'	6B	','	6C	'.'	6D	'/'	6E	SHIFTKEYS+ RIGHTSHIFT	6F	'\n'
70	RF(13)	71	STRING+ DOWNARROW	72	RF(15)	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
70	BUCKYBITS+ METABIT	71	' '	72	BUCKYBITS+ METABIT	73	HOLE	74	HOLE	75	HOLE	76	ERROR	77	IDLE

Shifted

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(11)	03	LF(2)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(11)
08	TF(3)	00	TF(12)	0A	TF(4)	0B	TF(13)	0C	TF(5)	0D	TF(14)	0E	TF(6)	0F	TF(15)
10	TF(7)	11	TF(8)	12	TF(9)	13	TF(10)	14	HOLE	15	RF(1)	16	RF(2)	17	RF(3)
18	HOLE	19	LF(3)	1A	LF(4)	1B	LF(12)	1C	HOLE	1D	c('[')	1E	'!'	1F	'@'
20	'#'	21	'$'	22	'%'	23	'^'	24	'&'	25	'*'	26	'('	27	')'
28	'_'	29	'+'	2A	'~'	2B	'\b'	2C	HOLE	2D	RF(4)	2E	RF(5)	2F	RF(6)
30	HOLE	31	LF(5)	32	LF(13)	33	LF(6)	34	HOLE	35	'\t'	36	'Q'	37	'W'
38	'E'	39	'R'	3A	'T'	3B	'Y'	3C	'U'	3D	'I'	3E	'O'	3F	'P'
40	'{'	41	'}'	42	0x7F	43	HOLE	44	RF(7)	45	STRING+ UPARROW	46	RF(9)	47	HOLE
48	LF(7)	49	LF(8)	4A	LF(14)	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	'A'	4E	'S'	4F	'D'
50	'F'	51	'G'	52	'H'	53	'J'	54	'K'	55	'L'	56	':'	57	'"'
58	'|'	59	'\r'	5A	HOLE	5B	STRING+ LEFTARROW	5C	RF(11)	5D	STRING+ RIGHTARROW	5E	HOLE	5F	LF(9)
60	LF(15)	61	LF(10)	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	'Z'	65	'X'	66	'C'	67	'V'
68	'B'	69	'N'	6A	'M'	6B	'<'	6C	'>'	6D	'?'	6E	SHIFTKEYS+ RIGHTSHIFT	6F	'\n'
70	RF(13)	71	STRING+ DOWNARROW	72	RF(15)	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	BUCKYBITS+ METABIT	79	' '	7A	BUCKYBITS+ METABIT	7B	HOLE	7C	HOLE	7D	HOLE	7E	ERROR	7F	IDLE

Sun-2 Keyboard

Caps Locked

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(11)	03	LF(2)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(11)
08	TF(3)	09	TF(12)	0A	TF(4)	0B	TF(13)	0C	TF(5)	0D	TF(14)	0E	TF(6)	0F	TF(15)
10	TF(7)	11	TF(8)	12	TF(9)	13	TF(10)	14	HOLE	15	RF(1)	16	RF(2)	17	RF(3)
18	HOLE	19	LF(3)	1A	LF(4)	1B	LF(12)	1C	HOLE	1D	c('[')	1E	'1'	1F	'2'
20	'3'	21	'4'	22	'5'	23	'6'	24	'7'	25	'8'	26	'9'	27	'0'
28	'-'	29	'='	2A	'''	2B	'\b'	2C	HOLE	2D	RF(4)	2E	RF(5)	2F	RF(6)
30	HOLE	31	LF(5)	32	LF(13)	33	LF(6)	34	HOLE	35	'\t'	36	'Q'	37	'W'
38	'E'	39	'R'	3A	'T'	3B	'Y'	3C	'U'	3D	'I'	3E	'O'	3F	'P'
40	'['	41	']'	42	0x7F	43	HOLE	44	RF(7)	45	STRING+ UPARROW	46	RF(9)	47	HOLE
48	LF(7)	49	LF(8)	4A	LF(14)	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	'A'	4E	'S'	4F	'D'
50	'F'	51	'G'	52	'H'	53	'J'	54	'K'	55	'L'	56	';'	57	'\''
58	'\'	59	'\r'	5A	HOLE	5B	STRING+ LEFTARROW	5C	RF(11)	5D	STRING+ RIGHTARROW	5E	HOLE	5F	LF(9)
60	LF(15)	61	LF(10)	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	'Z'	65	'X'	66	'C'	67	'V'
68	'B'	69	'N'	6A	'M'	6B	','	6C	'.'	6D	'/'	6E	SHIFTKEYS+ RIGHTSHIFT	6F	'\n'
70	RF(13)	71	STRING+ DOWNARROW	72	RF(15)	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	BUCKYBITS+ METABIT	79	' '	7A	BUCKYBITS+ METABIT	7B	HOLE	7C	HOLE	7D	HOLE	7E	ERROR	7F	IDLE

Controlled

KeyValue	KeyValue	KeyValue	KeyValue	Key	Value	KeyValue	Key	Value	KeyValue
00 HOLE	01 BUCKYBITS+	02 LF(11)	03 LF(2)	04	HOLE	05 TF(1)	06	TF(2)	07 TF(11)
08 TF(3)	09 TF(12)	0A TF(4)	0B TF(13)	0C	TF(5)	0D TF(14)	0E	TF(6)	0F TF(15)
10 TF(7)	11 TF(8)	12 TF(9)	13 TF(10)	14	HOLE	15 RF(1)	16	RF(2)	17 RF(3)
18 HOLE	19 LF(3)	1A LF(4)	1B LF(12)	1C	HOLE	1D c('[')	1E	'1'	1F c('@')
20 '3'	21 '4'	22 '5'	23 c('"')	24	'7'	25 '8'	26	'9'	27 '0'
28 c('_')	29 '='	2A c('~')	2B '\b'	2C	HOLE	2D RF(4)	2E	RF(5)	2F RF(6)
30 HOLE	31 LF(5)	32 LF(13)	33 LF(6)	34	HOLE	35 '\t'	36	c('q')	37 c('w')
38 c('e')	39 c('r')	3A c('t')	3B c('y')	3C	c('u')	3D c('i')	3E	c('o')	3F c('p')
40 c('[')	41 c(']')	42 0x7F	43 HOLE	44	RF(7)	45 STRING+ UPARROW	46	RF(9)	47 HOLE
48 LF(7)	49 LF(8)	4A LF(14)	4B HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D c('a')	4E	c('s')	4F c('d')
50 c('f')	51 c('g')	52 c('h')	53 c('j')	54	c('k')	55 c('l')	56	';'	57 '\''
58 c('\')	59 '\r'	5A HOLE	5B STRING+ LEFTARROW	5C	RF(11)	5D STRING+ RIGHTARROW	5E	HOLE	5F LF(9)
60 LF(15)	61 LF(10)	62 HOLE	63 SHIFTKEYS+ LEFTSHIFT	64	c('z')	65 c('x')	66	c('c')	67 c('v')
68 c('b')	69 c('n')	6A c('m')	6B ','	6C	'.'	6D c('_')	6E	SHIFTKEYS+ RIGHTSHIFT	6F '\n'
70 RF(13)	71 STRING+ DOWNARROW	72 RF(15)	73 HOLE	74	HOLE	75 HOLE	76	HOLE	77 HOLE
78 BUCKYBITS+ METABIT	79 c(' ')	7A BUCKYBITS+ METABIT	7B HOLE	7C	HOLE	7D HOLE	7E	ERROR	7F IDLE

Sun-2 Keyboard

Key Up

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	OOPS	03	OOPS	04	HOLE	05	OOPS	06	OOPS	07	OOPS
08	OOPS	09	OOPS	0A	OOPS	0B	OOPS	0C	OOPS	0D	OOPS	0E	OOPS	0F	OOPS
10	OOPS	11	OOPS	12	OOPS	13	OOPS	14	HOLE	15	OOPS	16	OOPS	17	NOP
18	HOLE	19	OOPS	1A	OOPS	1B	OOPS	1C	HOLE	1D	NOP	1E	NOP	1F	NOP
20	NOP	21	NOP	22	NOP	23	NOP	24	NOP	25	NOP	26	NOP	27	NOP
28	NOP	29	NOP	2A	NOP	2B	NOP	2C	HOLE	2D	OOPS	2E	OOPS	2F	NOP
30	HOLE	31	OOPS	32	OOPS	33	OOPS	34	HOLE	35	NOP	36	NOP	37	NOP
38	NOP	39	NOP	3A	NOP	3B	NOP	3C	NOP	3D	NOP	3E	NOP	3F	NOP
40	NOP	41	NOP	42	NOP	43	HOLE	44	OOPS	45	OOPS	46	NOP	47	HOLE
48	OOPS	49	OOPS	4A	OOPS	4B	HOLE	4C	SHIFTKEYS+ LEFTCTRL	4D	NOP	4e	NOP	4F	NOP
50	NOP	51	NOP	52	NOP	53	NOP	54	NOP	55	NOP	56	NOP	57	NOP
58	NOP	59	NOP	5A	HOLE	5B	OOPS	5C	OOPS	5D	NOP	5E	HOLE	5F	OOPS
60	OOPS	61	OOPS	62	HOLE	63	SHIFTKEYS+ LEFTSHIFT	64	NOP	65	NOP	66	NOP	67	NOP
68	NOP	69	NOP	6A	NOP	6B	NOP	6C	NOP	6D	NOP	6E	SHIFTKEYS+ RIGHTSHIFT	6F	NOP
70	OOPS	71	OOPS	72	NOP	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	BUCKYBITS+ METABIT	79	NOP	7A	BUCKYBITS+ METABIT	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	RESET

Micro Switch 103SD32-2 Keyboard

Unshifted

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(2)	03	LF(3)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(3)
08	TF(4)	09	TF(5)	0A	TF(6)	0B	TF(7)	0C	TF(8)	0D	TF(9)	0E	TF(10)	0F	TF(11)
10	TF(12)	11	TF(13)	12	TF(14)	13	c('[')	14	HOLE	15	RF(1)	16	'+'	17	'–'
18	HOLE	19	LF(4)	1A	'\f'	1B	LF(6)	1C	HOLE	1D	SHIFTKEYS+ CAPSLOCK	1E	'1'	1F	'2'
20	'3'	21	'4'	22	'5'	23	'6'	24	'7'	25	'8'	26	'9'	27	'0'
28	'–'	29	'='	2A	'''	2B	'\b'	2C	HOLE	2D	'7'	2E	'8'	2F	'9'
30	HOLE	31	LF(7)	32	STRING+ UPARROW	33	LF(9)	34	HOLE	35	'\t'	36	'q'	37	'w'
38	'e'	39	'r'	3A	't'	3B	'y'	3C	'u'	3D	'i'	3E	'o'	3F	'p'
40	'{'	41	'}'	42	'_'	43	HOLE	44	'4'	45	'5'	46	'6'	47	HOLE
48	STRING+ LEFTARROW	49	STRING+ HOMEARROW	4A	STRING+ RIGHTARROW	4B	HOLE	4C	SHIFTKEYS+ SHIFTLOCK	4D	'a'	4E	's'	4F	'd'
50	'f'	51	'g'	52	'h'	53	'j'	54	'k'	55	'l'	56	';'	57	':'
58	'|'	59	'\r'	5A	HOLE	5B	'1'	5C	'2'	5D	'3'	5E	HOLE	5F	NOSCROLL
60	STRING+ DOWNARROW	61	LF(15)	62	HOLE	63	HOLE	64	SHIFTKEYS+ LEFTSHIFT	65	'z'	66	'x'	67	'c'
68	'v'	69	'b'	6A	'n'	6B	'm'	6C	','	6D	'.'	6E	'/'	6F	SHIFTKEYS+ RIGHTSHIFT
70	NOP	71	0x7F	72	'0'	73	NOP	74	'.'	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	SHIFTKEYS+ LEFTCTRL	7B	' '	7C	SHIFTKEYS+ RIGHTCTRL	7D	HOLE	7E	HOLE	7F	IDLE

Micro Switch 103SD32-2 Keyboard

Shifted

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS SYSTEMBIT	02	LF(2)	03	LF(3)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(3)
08	TF(4)	09	TF(5)	0A	TF(6)	0B	TF(7)	0C	TF(8)	0D	TF(9)	0E	TF(10)	0F	TF(11)
10	TF(12)	11	TF(13)	12	TF(14)	13	c('[')	14	HOLE	15	RF(1)	16	'+'	17	'–'
18	HOLE	19	LF(4)	1A	'\f'	1B	LF(6)	1C	HOLE	1D	SHIFTKEYS+ CAPSLOCK	1E	'!'	1F	'"'
20	'#'	21	'$'	22	'%'	23	'&'	24	'\''	25	'('	26	')'	27	'0'
28	'='	29	'~'	2A	'@'	2B	'\b'	2C	HOLE	2D	'7'	2E	'8'	2F	'9'
30	HOLE	31	LF(7)	32	STRING+ UPARROW	33	LF(9)	34	HOLE	35	'\t'	36	'Q'	37	'W'
38	'E'	39	'R'	3A	'T'	3B	'Y'	3C	'U'	3D	'I'	3E	'O'	3F	'P'
40	'['	41	']'	42	'_'	43	HOLE	44	'4'	45	'5'	46	'6'	47	HOLE
48	STRING+ LEFTARROW	49	STRING+ HOMEARROW	4A	STRING+ RIGHTARROW	4B	HOLE	4C	SHIFTKEYS+ SHIFTLOCK	4D	'A'	4E	'S'	4F	'D'
50	'F'	51	'G'	52	'H'	53	'J'	54	'K'	55	'L'	56	'+'	57	'*'
58	'\'	59	'\r'	5A	HOLE	5B	'1'	5C	'2'	5D	'3'	5E	HOLE	5F	NOSCROLL
60	STRING+ DOWNARROW	61	LF(15)	62	HOLE	63	HOLE	64	SHIFTKEYS+ LEFTSHIFT	65	'Z'	66	'X'	67	'C'
68	'V'	69	'B'	6A	'N'	6B	'M'	6C	'<'	6D	'>'	6E	'?'	6F	SHIFTKEYS+ RIGHTSHIFT
70	NOP	71	0x7F	72	'0'	73	NOP	74	'.'	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	SHIFTKEYS+ RIGHTSHIFT	7B	' '	7C	SHIFTKEYS+ LEFTCTRL	7D	HOLE	7E	HOLE	7F	IDLE

Caps Locked

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(2)	03	LF(3)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(3)
08	TF(4)	09	TF(5)	0A	TF(6)	0B	TF(7)	0C	TF(8)	0D	TF(9)	0E	TF(10)	0F	TF(11)
10	TF(12)	11	TF(13)	12	TF(14)	13	c('[')	14	HOLE	15	RF(1)	16	'+'	17	'–'
18	HOLE	19	LF(4)	1A	'\f'	1B	LF(6)	1C	HOLE	1D	SHIFTKEYS+ CAPSLOCK	1E	'1'	1F	'2'
20	'3'	21	'4'	22	'5'	23	'6'	24	'7'	25	'8'	26	'9'	27	'0'
28	'–'	29	'~'	2A	'`'	2B	'\b'	2C	HOLE	2D	'7'	2E	'8'	2F	'9'
30	HOLE	31	LF(7)	32	STRING+ UPARROW	33	LF(9)	34	HOLE	35	'\t'	36	'Q'	37	'W'
38	'E'	39	'R'	3A	'T'	3B	'Y'	3C	'U'	3D	'I'	3E	'O'	3F	'P'
40	'{'	41	'}'	42	'_'	43	HOLE	44	'4'	45	'5'	46	'6'	47	HOLE
48	STRING+ LEFTARROW	49	STRING+ HOMEARROW	4A	STRING+ RIGHTARROW	4B	HOLE	4C	SHIFTKEYS+ SHIFTLOCK	4D	'A'	4E	'S'	4F	'D'
50	'F'	51	'G'	52	'H'	53	'J'	54	'K'	55	'L'	56	';'	57	':'
58	'|'	59	'\r'	5A	HOLE	5B	'1'	5C	'2'	5D	'3'	5E	HOLE	5F	NOSCROLL
60	STRING+ DOWNARROW	61	LF(15)	62	HOLE	63	HOLE	64	SHIFTKEYS+ LEFTSHIFT	65	'Z'	66	'X'	67	'C'
68	'V'	69	'B'	6A	'N'	6B	'M'	6C	','	6D	'.'	6E	'/'	6F	SHIFTKEYS+ RIGHTSHIFT
70	NOP	71	0x7F	72	'0'	73	NOP	74	'.'	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	SHIFTKEYS+ LEFTCTRL	7B	' '	7C	SHIFTKEYS RIGHTCTRL	7D	HOLE	7E	HOLE	7F	IDLE

Micro Switch 103SD32-2 Keyboard

Controlled

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	LF(2)	03	LF(3)	04	HOLE	05	TF(1)	06	TF(2)	07	TF(3)
08	TF(4)	09	TF(5)	0A	TF(6)	0B	TF(7)	0C	TF(8)	0D	TF(9)	0E	TF(10)	0F	TF(11)
10	TF(02)	11	TF(03)	12	TF(04)	13	c('[')	14	HOLE	15	RF(0)	16	OOPS	17	OOPS
18	HOLE	19	LF(4)	1A	'\f'	1B	LF(6)	1C	HOLE	1D	SHIFTKEYS+ CAPSLOCK	1E	OOPS	1F	OOPS
20	OOPS	21	OOPS	22	OOPS	23	OOPS	24	OOPS	25	OOPS	26	OOPS	27	OOPS
28	OOPS	29	c('^')	2A	c('@')	2B	'\b'	2C	HOLE	2D	OOPS	2E	OOPS	2F	OOPS
30	HOLE	31	LF(7)	32	STRING+ UPARROW	33	F(9)	34	HOLE	35	'\t'	36	CTRLQ	37	c('W')
38	c('E')	39	c('R')	3A	c('T')	3B	c('Y')	3C	c('U')	3D	c('I')	3E	c('O')	3F	c('P')
40	c('[')	41	c(']')	42	c('_')	43	HOLE	44	OOPS	45	OOPS	46	OOPS	47	HOLE
48	STRING+ LEFTARROW	49	STRING+ HOMEARROW	4A	STRING+ RIGHTARROW	4B	HOLE	4C	SHIFTKEYS+ SHIFTLOCK	4D	c('A')	4E	CTRLS	4F	c('D')
50	c('F')	51	c('G')	52	c('H')	53	c('J')	54	c('K')	55	c('L')	56	OOPS	57	OOPS
58	c('\')	59	'\r'	5A	HOLE	5B	OOPS	5C	OOPS	5D	OOPS	5E	HOLE	5F	NOSCROLL
60	STRING+ DOWNARROW	61	LF(15)	62	HOLE	63	HOLE	64	SHIFTKEYS+ LEFTSHIFT	65	c('Z')	66	c('X')	67	c('C')
68	c('V')	69	c('B')	6A	c('N')	6B	c('M')	6C	OOPS	6D	OOPS	6E	OOPS	6F	SHIFTKEYS+ RIGHTSHIFT
70	NOP	71	0x7F	72	OOPS	73	NOP	74	OOPS	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	SHIFTKEYS+ LEFTCTRL	7B	'\0'	7C	SHIFTKEYS+ RIGHTCTRL	7D	HOLE	7E	HOLE	7F	IDLE

Key Up

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	OOPS	03	OOPS	04	HOLE	05	OOPS	06	OOPS	07	OOPS
08	OOPS	09	OOPS	0A	OOPS	0B	OOPS	0C	OOPS	0D	OOPS	0E	OOPS	0F	OOPS
10	OOPS	11	OOPS	12	OOPS	13	NOP	14	HOLE	15	OOPS	16	NOP	17	NOP
18	HOLE	19	OOPS	1A	NOP	1B	OOPS	1C	HOLE	1D	SHIFTKEYS+ CAPSLOCK	1E	NOP	1F	NOP
20	NOP	21	NOP	22	NOP	23	NOP	24	NOP	25	NOP	26	NOP	27	NOP
28	NOP	29	NOP	2A	NOP	2B	NOP	2C	HOLE	2D	NOP	2E	NOP	2F	NOP
30	HOLE	31	OOPS	32	NOP	33	OOPS	34	HOLE	35	NOP	36	NOP	37	NOP
38	NOP	39	NOP	3A	NOP	3B	NOP	3C	NOP	3D	NOP	3E	NOP	3F	NOP
40	NOP	41	NOP	42	NOP	43	HOLE	44	NOP	45	NOP	46	NOP	47	HOLE
48	NOP	49	NOP	4A	NOP	4B	HOLE	4C	SHIFTKEYS+ SHIFTLOCK	4D	NOP	4E	NOP	4F	NOP
50	NOP	51	NOP	52	NOP	53	NOP	54	NOP	55	NOP	56	NOP	57	NOP
58	NOP	59	NOP	5A	HOLE	5B	NOP	5C	NOP	5D	NOP	5E	HOLE	5F	NOP
60	NOP	61	OOPS	62	HOLE	63	HOLE	64	SHIFTKEYS+ LEFTSHIFT	65	NOP	66	NOP	67	NOP
68	NOP	69	NOP	6A	NOP	6B	NOP	6C	NOP	6D	NOP	6E	NOP	6F	SHIFTKEYS+ RIGHTSHIFT
70	NOP	71	NOP	72	NOP	73	NOP	74	NOP	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	SHIFTKEYS+ LEFTCTRL	7B	NOP	7C	SHIFTKEYS+ RIGHTCTRL	7D	HOLE	7E	HOLE	7F	RESET

VT100-Style Keyboard

Unshifted

Key	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	HOLE	04	HOLE	05	HOLE	06	HOLE	07	HOLE
08	HOLE	09	HOLE	0A	STRING+ UPARROW	0B	STRING+ DOWNARROW	0C	STRING+ LEFTARROW	0D	STRING+ RIGHTARROW	0E	HOLE	0F	TF(1)
10	TF(2)	11	TF(3)	12	TF(4)	13	c('[')	14	'1'	15	'2'	16	'3'	17	'4'
18	'5'	19	'6'	1A	'7'	1B	'8'	1C	'9'	1D	'0'	1E	'–'	1F	'='
20	'`'	21	c('H')	22	BUCKYBITS+ METABIT	23	'7'	24	'8'	25	'9'	26	'–'	27	'\t'
28	'q'	29	'w'	2A	'e'	2B	'r'	2C	't'	2D	'y'	2E	'u'	2F	'i'
30	'o'	31	'p'	32	'['	33	']'	34	0x7F	35	'4'	36	'5'	37	'6'
38	','	39	SHIFTKEYS+ LEFTCTRL	3A	SHIFTKEYS+ CAPSLOCK	3B	'a'	3C	's'	3D	'd'	3E	'f'	3F	'g'
40	'h'	41	'j'	42	'k'	43	'l'	44	';'	45	'\''	46	'\r'	47	'\'
48	'1'	49	'2'	4A	'3'	4B	NOP	4C	NOSCROLL	4D	SHIFTKEYS+ LEFTSHIFT	4E	'z'	4F	'x'
50	'c'	51	'v'	52	'b'	53	'n'	54	'm'	55	','	56	'.'	57	'/'
58	SHIFTKEYS+ RIGHTSHIFT	59	'\n'	5A	'0'	5B	HOLE	5C	'.'	5D	'\r'	5E	HOLE	5F	HOLE
60	HOLE	61	HOLE	62	' '	63	HOLE	64	HOLE	65	HOLE	66	HOLE	67	HOLE
68	HOLE	69	HOLE	6A	HOLE	6B	HOLE	6C	HOLE	6D	HOLE	6E	HOLE	6F	HOLE
70	HOLE	71	HOLE	72	HOLE	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	HOLE	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	IDLE

Shifted

Key	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	HOLE	04	HOLE	05	HOLE	06	HOLE	07	HOLE
08	HOLE	09	HOLE	0A	STRING+ UPARROW	0B	STRING+ DOWNARROW	0C	STRING+ LEFTARROW	0D	STRING+ RIGHTARROW	0E	HOLE	0F	TF(1)
10	TF(2)	11	TF(3)	12	TF(4)	13	c('[')	14	'!'	15	'@'	16	'#'	17	'$'
18	'%'	19	'^'	1A	'&'	1B	'*'	1C	'('	1D	')'	1E	'_'	1F	'+'
20	'~'	21	c('H')	22	BUCKYBITS+ METABIT	23	'7'	24	'8'	25	'9'	26	'–'	27	'\t'
28	'Q'	29	'W'	2A	'E'	2B	'R'	2C	'T'	2D	'Y'	2E	'U'	2F	'I'
30	'O'	31	'P'	32	'{'	33	'}'	34	0x7F	35	'4'	36	'5'	37	'6'
38	','	39	SHIFTKEYS+ LEFTCTRL	3A	SHIFTKEYS+ CAPSLOCK	3B	'A'	3C	'S'	3D	'D'	3E	'F'	3F	'G'
40	'H'	41	'J'	42	'K'	43	'L'	44	':'	45	'"'	46	'\r'	47	'|'
48	'1'	49	'2'	4A	'3'	4B	NOP	4C	NOSCROLL	4D	SHIFTKEYS+ LEFTSHIFT	4E	'Z'	4F	'X'
50	'C'	51	'V'	52	'B'	53	'N'	54	'M'	55	'<'	56	'>'	57	'?'
58	SHIFTKEYS+ RIGHTSHIFT	59	'\n'	5A	'0'	5B	HOLE	5C	'.'	5D	'\r'	5E	HOLE	5F	HOLE
60	HOLE	61	HOLE	62	' '	63	HOLE	64	HOLE	65	HOLE	66	HOLE	67	HOLE
68	HOLE	69	HOLE	6A	HOLE	6B	HOLE	6C	HOLE	6D	HOLE	6E	HOLE	6F	HOLE
70	HOLE	71	HOLE	72	HOLE	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	HOLE	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	IDLE

VT100-Style Keyboard

Caps Locked

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	HOLE	04	HOLE	05	HOLE	06	HOLE	07	HOLE
08	HOLE	09	HOLE	0A	STRING+ UPARROW	0B	STRING+ DOWNARROW	0C	STRING+ LEFTARROW	0D	STRING+ RIGHTARROW	0E	HOLE	0F	TF(1)
10	TF(2)	11	TF(3)	12	TF(4)	13	c('[')	14	'1'	15	'2'	16	'3'	17	'4'
18	'5'	19	'6'	1A	'7'	1B	'8'	1C	'9'	1D	'0'	1E	'-'	1F	'='
20	'`'	21	c('H')	22	BUCKYBITS+ METABIT	23	'7'	24	'8'	25	'9'	26	'-'	27	'\t'
28	'Q'	29	'W'	2A	'E'	2B	'R'	2C	'T'	2D	'Y'	2E	'U'	2F	'I'
30	'O'	31	'P'	32	'['	33	']'	34	0x7F	35	'4'	36	'5'	37	'6'
38	','	39	SHIFTKEYS+ LEFTCTRL	3A	SHIFTKEYS+ CAPSLOCK	3B	'A'	3C	'S'	3D	'D'	3E	'F'	3F	'G'
40	'H'	41	'J'	42	'K'	43	'L'	44	';'	45	'\''	46	'\r'	47	'\'
48	'1'	49	'2'	4A	'3'	4B	NOP	4C	NOSCROLL	4D	SHIFTKEYS+ LEFTSHIFT	4E	'Z'	4F	'X'
50	'C'	51	'V'	52	'B'	53	'N'	54	'M'	55	','	56	'.'	57	'/'
58	SHIFTKEYS+ RIGHTSHIFT	59	'\n'	5A	'0'	5B	HOLE	5C	'.'	5D	'\r'	5E	HOLE	5F	HOLE
60	HOLE	61	HOLE	62	' '	63	HOLE	64	HOLE	65	HOLE	66	HOLE	67	HOLE
68	HOLE	69	HOLE	6A	HOLE	6B	HOLE	6C	HOLE	6D	HOLE	6E	HOLE	6F	HOLE
70	HOLE	71	HOLE	72	HOLE	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	HOLE	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	IDLE

Controlled

Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value	Key	Value
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	HOLE	04	HOLE	05	HOLE	06	HOLE	07	HOLE
08	HOLE	09	HOLE	0A	STRING+ UPARROW	0B	STRING+ DOWNARROW	0C	STRING+ LEFTARROW	0D	STRING+ RIGHTARROW	0E	HOLE	0F	TF(1)
10	TF(2)	11	TF(3)	12	TF(4)	13	c('[')	14	'1'	15	c('@')	16	'3'	17	'4'
18	'5'	19	c('^')	1A	'7'	1B	'8'	1C	'9'	1D	'0'	1E	c('_')	1F	'='
20	c('^')	21	c('H')	22	BUCKYBITS+ METABIT	23	'7'	24	'8'	25	'9'	26	'-'	27	'\t'
28	CTRLQ	29	c('W')	2A	c('E')	2B	c('R')	2C	c('T')	2D	c('Y')	2E	c('U')	2F	c('I')
30	c('O')	31	c('P')	32	c('[')	33	c(']')	34	0x7F	35	'4'	36	'5'	37	'6'
38	','	39	SHIFTKEYS+ LEFTCTRL	3A	SHIFTKEYS+ CAPSLOCK	3B	c('A')	3C	CTRLS	3D	c('D')	3E	c('F')	3F	c('G')
40	c('H')	41	c('J')	42	c('K')	43	c('L')	44	':'	45	'"'	46	'\r'	47	c('\')
48	'1'	49	'2'	4A	'3'	4B	NOP	4C	NOSCROLL	4D	SHIFTKEYS+ LEFTSHIFT	4E	c('Z')	4F	c('X')
50	c('C')	51	c('V')	52	c('B')	53	c('N')	54	c('M')	55	','	56	'.'	57	c('_')
58	SHIFTKEYS+ RIGHTSHIFT	59	'\n'	5A	'0'	5B	HOLE	5C	'.'	5D	HOLE	5E	HOLE	5F	HOLE
60	HOLE	61	HOLE	62	c(' ')	63	HOLE	64	HOLE	65	HOLE	66	HOLE	67	HOLE
68	HOLE	69	HOLE	6A	HOLE	6B	HOLE	6C	HOLE	6D	HOLE	6E	HOLE	6F	HOLE
70	HOLE	71	HOLE	72	HOLE	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	HOLE	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	IDLE

VT100-Style Keyboard
Key Up

Key	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*	*Key*	*Value*
00	HOLE	01	BUCKYBITS+ SYSTEMBIT	02	HOLE	03	HOLE	04	HOLE	05	HOLE	06	HOLE	07	HOLE
08	HOLE	09	HOLE	0A	NOP	0B	NOP	0C	NOP	0D	NOP	0E	HOLE	0F	OOPS
10	OOPS	11	OOPS	12	OOPS	13	NOP	14	NOP	15	NOP	16	NOP	17	NOP
18	NOP	19	NOP	1A	NOP	1B	NOP	1C	NOP	1D	NOP	1E	NOP	1F	NOP
20	NOP	21	NOP	22	BUCKYBITS+ METABIT	23	NOP	24	NOP	25	NOP	26	NOP	27	NOP
28	NOP	29	NOP	2A	NOP	2B	NOP	2C	NOP	2D	NOP	2E	NOP	2F	NOP
30	NOP	31	NOP	32	NOP	33	NOP	34	NOP	35	NOP	36	NOP	37	NOP
38	NOP	39	SHIFTKEYS+ LEFTCTRL	3A	SHIFTKEYS+ CAPSLOCK	3B	NOP	3C	NOP	3D	NOP	3E	NOP	3F	NOP
40	NOP	41	NOP	42	NOP	43	NOP	44	NOP	45	NOP	46	NOP	47	NOP
48	NOP	49	NOP	4A	NOP	4B	NOP	4C	NOP	4D	SHIFTKEYS+ LEFTSHIFT	4E	NOP	4F	NOP
50	NOP	51	NOP	52	NOP	53	NOP	54	NOP	55	NOP	56	NOP	57	NOP
58	SHIFTKEYS+ RIGHTSHIFT	59	NOP	5A	NOP	5B	HOLE	5C	NOP	5D	NOP	5E	HOLE	5F	HOLE
60	HOLE	61	HOLE	62	NOP	63	HOLE	64	HOLE	65	HOLE	66	HOLE	67	HOLE
68	HOLE	69	HOLE	6A	HOLE	6B	HOLE	6C	HOLE	6D	HOLE	6E	HOLE	6F	HOLE
70	HOLE	71	HOLE	72	HOLE	73	HOLE	74	HOLE	75	HOLE	76	HOLE	77	HOLE
78	HOLE	79	HOLE	7A	HOLE	7B	HOLE	7C	HOLE	7D	HOLE	7E	HOLE	7F	RESET

FILES

/dev/kbd

SEE ALSO

setkeys(1), click(1)

The SunView System Programmer's Guide - Appendix: Writing a Virtual User Input Device Driver (describes *Firm_event* format)

NAME

le – Sun-3/50 10 Mb/s Ethernet interface

SYNOPSIS

device le0 at obio ? csr

DESCRIPTION

The *le* interface provides access to a 10 Mb/s Ethernet network through a Sun-3 controller using the AMD LANCE (Local Area Network Controller for Ethernet) Am7990 chip. For a general description of network interfaces see *if*(4N).

The synopsis line above specifies the first and only Ethernet controller on a Sun-3/50.

DIAGNOSTICS

le%d: transmitter frozen -- resetting A bug in the LANCE chip has caused the chip's transmitter section to stop. The driver has detected this condition and reinitialized the chip.

le%d: out of mbufs: output packet dropped The driver has run out of memory to use to buffer packets on output. The packet being transmitted at the time of occurrence is lost. This error is usually symptomatic of trouble elsewhere in the kernel.

le%d: stray transmitter interrupt The LANCE chip has signalled that it completed transmitting a packet but the driver has sent no such packet.

le%d: LANCE Rev C/D Extra Byte(s) bug; Packet dropped The LANCE chip's internal silo pointers have become misaligned. This error arises from a chip bug.

le%d: trailer error An incoming packet claimed to have a trailing header but did not.

le%d: runt packet An incoming packet's size was below the Ethernet minimum transmission size.

le%d: Receive buffer error - BUFF bit set in rmd This error "should never happen," as it occurs only in conjunction with a LANCE feature that the driver does not use.

le%d: Received packet with STP bit in rmd cleared This error "should never happen," as it occurs only in conjunction with a LANCE feature that the driver does not use.

le%d: Received packet with ENP bit in rmd cleared This error "should never happen," as it occurs only in conjunction with a LANCE feature that the driver does not use.

le%d: Transmit buffer error - BUFF bit set in tmd Excessive bus contention has prevented the LANCE chip from gathering packet contents quickly enough to sustain the packet's transmission over the Ethernet. The affected packet is lost.

le%d: Transmit late collision - Net problem? A packet collision has occurred after the channel's slot time has elapsed. This error usually indicates faulty hardware elsewhere on the net.

le%d: No carrier - transceiver cable problem? The LANCE chip has lost input to its carrier detect pin while trying to transmit a packet.

le%d: Transmit retried more than 16 times - net jammed Network activity has become so intense that sixteen successive transmission attempts failed, causing the LANCE chip to give up on the current packet.

le%d: missed packet The driver has dropped an incoming packet because it had no buffer space for it.

le%d: Babble error - sent a packet longer than the maximum length While transmitting a packet, the LANCE chip has noticed that the packet's length exceeds the maximum allowed for Ethernet. This error indicates a kernel bug.

le%d: Memory Error! Ethernet chip memory access timed out The LANCE chip timed out while trying to acquire the bus for a DVMA transfer.

le%d: Reception stopped Because of some other error, the receive section of the LANCE chip shut down and had to be restarted.

le%d: Transmission stopped Because of some other error, the transmit section of the LANCE chip shut down and had to be restarted.

NAME

lo – software loopback network interface

SYNOPSIS

pseudo-device loop

DESCRIPTION

The *loop* device is a software loopback network interface; see *if*(4N) for a general description of network interfaces.

The *loop* interface is used for performance analysis and software testing, and to provide guaranteed access to Internet protocols on machines with no local network interfaces. A typical application is the *comsat*(8C) server which accepts notification of mail delivery through a particular port on the loopback interface.

By default, the loopback interface is accessible at Internet address 127.0.0.1 (non-standard); this address may be changed with the SIOCSIFADDR ioctl.

DIAGNOSTICS

lo%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

if(4N), inet(4F)

BUGS

It should handle all address and protocol families. An approved network address should be reserved for this interface.

NAME

mem, kmem, vme16d16, vme24d16, vme32d16, vme16d32, vme24d32, vme32d32, mbmem, mbio, – main memory and bus I/O space

SYNOPSIS

None; included with standard system.

DESCRIPTION

These devices are special files that map memory and bus I/O space. They may be read, written, seek'ed and (except for kmem) *mmap*(2)'ed.

Mem is a special file that is an image of the physical memory of the computer. It may be used, for example, to examine (and even to patch) the system.

Kmem is a special file that is an image of the kernel virtual memory of the system.

vme16d16 (also known as *vme16*) is a special file that is an image of VMEbus 16-bit addresses with 16-bit data. *Vme16* address space extends from 0 to 64K.

vme24d16 (also known as *vme24*) is a special file that is an image of VMEbus 24-bit addresses with 16-bit data. *Vme24* address space extends from 0 to 16 Megabytes. The VME 16-bit address space overlaps the top 64K of the 24-bit address space.

SUN-3 VMEBUS ONLY

vme32d16 is a special file that is an image of VMEbus 32-bit addresses with 16-bit data.

vme16d32 is a special file that is an image of VMEbus 16-bit addresses with 32-bit data.

vme24d32 is a special file that is an image of VMEbus 24-bit addresses with 32-bit data.

vme32d32 (also known as *vme32*) is a special file that is an image of VMEbus 32-bit addresses with 32-bit data. *Vme32* address space extends from 0 to 4 Giggabytes. The VME 24-bit address space overlaps the top 16 Megabytes of the 32-bit address space.

*vme** type special files can only be accessed in VME based systems.

SUN-2 MULTIBUS ONLY

Mbmem is a special file that is an image of the Multibus memory of the system. Multibus memory is in the range from 0 to 1 Megabyte. *Mbmem* can only be accessed in Multibus based systems.

Mbio is a special file that is an image of the Multibus I/O space. Multibus I/O space extends from 0 to 64K. *Mbio* can only be accessed in Multibus based systems.

When reading and writing *mbmem* and *mbio* odd counts or offsets cause byte accesses and even counts and offsets cause word accesses.

FILES

/dev/mem
/dev/kmem
/dev/mbmem
/dev/mbio
/dev/vme16d16
/dev/vme16
/dev/vme24d16
/dev/vme24
/dev/vme32d16
/dev/vme16d32
/dev/vme24d32
/dev/vme32d32
/dev/vme32

NAME

mouse – Sun mouse

SYNOPSIS

pseudo-device ms3

DESCRIPTION

The *mouse* interface provides access to the Sun Workstation mouse.

The mouse incorporates a microprocessor which generates a byte-stream protocol encoding mouse motions.

Each mouse sample in the byte stream consists of three bytes: the first byte gives the button state with value 0x87|~*but*, where *but* is the low three bits giving the mouse buttons, where a 0 (zero) bit means that a button is pressed, and a 1 (one) bit means a button is not pressed. Thus if the left button is down the value of this sample is 0x83, while if the right button is down the byte is 0x86.

The next two bytes of each sample give the *x* and *y* delta's of this sample as signed bytes. The mouse uses a lower-left coordinate system, so moves to the right on the screen yield positive *x* values and moves down the screen yield negative *y* values.

The beginning of a sample is identifiable because the delta's are constrained to not have values in the range 0x80-0x87.

The mouse can be used as a device that emits *Firm_events* as specified by the protocol of a *Virtual User Input Device*. It understands VUIDSFORMAT, VUIDGFORMAT, VUIDSADDR and VUIDGADDR ioctls (see reference below).

FILES

/dev/mouse

SEE ALSO

win(4S)

The SunView System Programmer's Guide

NAME

mti – Systech MTI-800/1600 multi-terminal interface

SYNOPSIS — SUN-3

device mti0 at vme16d16 ? csr 0x620 flags 0xffff priority 4 vector mtiintr 0x88
device mti1 at vme16d16 ? csr 0x640 flags 0xffff priority 4 vector mtiintr 0x89
device mti2 at vme16d16 ? csr 0x660 flags 0xffff priority 4 vector mtiintr 0x8a
device mti3 at vme16d16 ? csr 0x680 flags 0xffff priority 4 vector mtiintr 0x8b

SYNOPSIS — SUN-2

device mti0 at mbio ? csr 0x620 flags 0xffff priority 4
device mti1 at mbio ? csr 0x640 flags 0xffff priority 4
device mti2 at mbio ? csr 0x660 flags 0xffff priority 4
device mti3 at mbio ? csr 0x680 flags 0xffff priority 4
device mti0 at vme16 ? csr 0x620 flags 0xffff priority 4 vector mtiintr 0x88
device mti1 at vme16 ? csr 0x640 flags 0xffff priority 4 vector mtiintr 0x89
device mti2 at vme16 ? csr 0x660 flags 0xffff priority 4 vector mtiintr 0x8a
device mti3 at vme16 ? csr 0x680 flags 0xffff priority 4 vector mtiintr 0x8b

DESCRIPTION

The Systech MTI card provides 8 (MTI-800) or 16 (MTI-1600) serial communication lines with modem control. Each line behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Bit *i* of flags may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying "flags 0x0004" in the specification of mti0 would cause line tty02 to be treated in this way.

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 0 – 127 correspond directly to the normal tty lines and are named *tty**. Minor device numbers in the range 128 – 256 correspond to the same physical lines as those above (i.e. the same line as the minor device number minus 128) and are (conventionally) named *cua**. The *cua* lines are special in that they can be opened even when there is no carrier on the line. Once a *cua* line is opened, the corresponding tty line can not be opened until the *cua* line is closed. Also, if the *tty* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding *cua* line can not be opened. This allows a modem to be attached to */dev/tty00* (usually renamed to */dev/ttyd0*) and used for dialin (by enabling the line for login in */etc/ttys*) and also used for dialout (by *tip*(1C) or *uucp*(1C)) as */dev/cua0* when no one is logged in on the line. Note that the bit in the flags word in the config file (see above) must be zero for this line.

WIRING

The Systech requires the CTS modem control signal to operate. If the device does not supply CTS then RTS should be jumpered to CTS at the distribution panel (short pins 4 to 5). Also, the CD (carrier detect) line does not work properly. When connecting a modem, the modem's CD line should be wired to DSR, which the software will treat as carrier detect.

FILES

/dev/tty0[0-9a-f] hardwired tty lines
/dev/ttyd[0-9a-f] dialin tty lines
/dev/cua[0-9a-f] dialout tty lines

SEE ALSO

tty(4), zs(4S)

DIAGNOSTICS

Most of these diagnostics "should never happen" and their occurrence usually indicates problems elsewhere in the system.

mti*n*,*n*: silo overflow.
More than 512 characters have been received by the mti hardware without being read by the

software. Extremely unlikely to occur.

mti*n*: error *n*.
The mti returned the indicated error code. See the mti manual.

mti*n*: DMA output error.
The mti encountered an error while trying to do DMA output.

mti*n*: impossible response *n*.
The mti returned an error it couldn't understand.

NAME

mtio – UNIX system magnetic tape interface

SYNOPSIS

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

DESCRIPTION

The files *mt0, ..., mt15* refer to the UNIX system magnetic tape drives, which read and write magnetic tape in 2048 byte blocks (the 2048 is actually BLKDEV_IOSIZE in <sys/param.h>). The following description applies to any of the transport/controller pairs. The files *mt0, ..., mt3* and *mt8, ..., mt11* are rewound when closed; the others are not. When a nine track tape file, open for writing or just written, is closed, two end-of-files are written; if the tape is not to be rewound it is positioned with the head between the two tape-marks. When a 1/4" tape file, (due to a bug, only if) just written, is closed, only one end of file mark is written because of the inability to overwrite data on a 1/4" tape; see below.

1/4" tapes are not able to back up and always write fixed sized blocks. Since they cannot back up, they cannot support backward space file and backward space record. Since they always write fixed sized blocks, the size of transfers using the raw interface (see below) must be a multiple of the underlying block-size, usually 512 bytes.

1/4" tapes also have an unusual tape format. They have parallel tracks, but only record information on one track at a time, switching to another track near the physical end of the medium. They erase all the tracks at once while writing the first track. Therefore, they cannot, in general, overwrite previously written data. If the old data were not on the first track, it would not be erased before being overwritten, and the result would be unreadable.

The *mt* files discussed above are useful when it you want to access the tape in a way compatible with ordinary files. When using foreign tapes, and especially when reading or writing long records, the 'raw' interface is appropriate. The associated files are named *rmt0, ..., rmt15*, but the same minor-device considerations as for the regular files still apply. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size. In raw tape I/O seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

A number of additional ioctl operations are available on raw magnetic tape. The following definitions are from <sys/mtio.h>:

```
/*
 * Structures and definitions for mag tape I/O control commands
 */

/* structure for MTIOCTOP - mag tape op command */
struct    mtop    {
          short     mt_op;              /* operations defined below */
          daddr_t   mt_count;           /* how many of them */
};

/* operations */
#define   MTWEOF      0           /* write an end-of-file record */
#define   MTFSF 1               /* forward space file */
#define   MTBSF 2               /* backward space file */
#define   MTFSR 3               /* forward space record */
#define   MTBSR 4               /* backward space record */
#define   MTREW       5           /* rewind */
#define   MTOFFL      6           /* rewind and put the drive offline */
#define   MTNOP       7           /* no operation, sets status only */
```

```
#define  MTRETEN     8          /* retension the tape */
#define  MTERASE     9          /* erase the entire tape */

/* structure for MTIOCGET - mag tape get status command */

struct   mtget   {
         short    mt_type;      /* type of magtape device */
         /* the following two registers are grossly device dependent */
         short    mt_dsreg;     /* "drive status" register */
         short    mt_erreg;     /* "error" register */
         /* end device-dependent registers */
         short    mt_resid;     /* residual count */
         /* the following two are not yet implemented */
         daddr_t  mt_fileno;    /* file number of current position */
         daddr_t  mt_blkno;     /* block number of current position */
         /* end not yet implemented */
};

/*
 * Constants for mt_type byte
 */
#define  MT_ISTS     0x01       /* vax: unibus ts-11 */
#define  MT_ISHT     0x02       /* vax: massbus tu77, etc */
#define  MT_ISTM     0x03       /* vax: unibus tm-11 */
#define  MT_ISMT     0x04       /* vax: massbus tu78 */
#define  MT_ISUT     0x05       /* vax: unibus gcr */
#define  MT_ISCPC    0x06       /* sun: Multibus tapemaster */
#define  MT_ISAR     0x07       /* sun: Multibus archive */
#define  MT_ISSC     0x08       /* sun: SCSI archive */
#define  MT_ISXY     0x09       /* sun: Xylogics 472 */

/* mag tape io control commands */
#define  MTIOCTOP    _IOW(m, 1, struct mtop)  /* do a mag tape op */
#define  MTIOCGET    _IOR(m, 2, struct mtget) /* get tape status */

#ifndef KERNEL
#define  DEFTAPE     "/dev/rmt12"
#endif
```

FILES

/dev/mt*
/dev/rmt*
/dev/rar*

SEE ALSO

mt(1), tar(1), ar(4S), tm(4S), st(4S), xt(4S)

NAME

nd – network disk driver

SYNOPSIS

pseudo-device nd

DESCRIPTION

The network disk device, */dev/nd**, allows a client workstation to perform disk I/O operations on a server system over the network. To the client system, this device looks like any normal disk driver: it allows read/write operations at a given block number and byte count. Note that this provides a network *disk block* access service rather than a network *file* access service.

Typically the client system will have no disks at all. In this case */dev/nd0* contains the client's root file system (including */usr* files), and *nd1* is used as a paging area. Client access to these devices is converted to *net disk protocol* requests and sent to the server system over the network. The server receives the request, performs the actual disk I/O, and sends a response back to the client.

The server contains a table which lists the net address of each of his clients and the server disk partition which corresponds to each client unit number (nd0,1,...). This table resides in the server kernel in a structure owned by the nd device. The table is initialized by running the program */etc/nd* with text file */etc/nd.local* as its input. */etc/nd* then issues *ioctl*(2) functions to load the table into the kernel.

In addition to the read/write units */dev/nd**, there are *public* read-only units which are named */dev/ndp**. The correspondence to server partitions is specified by the */etc/nd.local* text file, in a similar manner to the private partitions. The public units can be used to provide shared access to binaries or libraries (*/bin*, */usr/bin*, */usr/ucb*, */usr/lib*) so that each diskless client does not have to consume space in his private partitions for these files. This is done by providing a public file system at the server (*/dev/ndp0*) which is mounted on */pub* of each diskless client. The clients then use symbolic links to read the public files: */bin -> /pub/bin*, */usr/ucb -> /pub/usr/ucb*. One requirement in this case is that the server (who has read/write access to this file system) should not perform write activity with any public filesystem. This is because each client is locally cacheing blocks, and may get out of sync with the physical disk image. In certain cases, the client will detect an inconsistency and panic.

One last type of unit is provided for use by the server. These are called *local* units and are named */dev/ndl**. The Sun physical disk sector 0 label only provides a limited number of partitions per physical disk (eight). Since this number is small and these partitions have somewhat fixed meanings, the nd driver itself has a *subpartitioning* capability built-in. This allows the large server physical disk partition (e.g. */dev/xy0g*) to be broken up into any number of diskless client partitions. Of course on the client side these would be referenced as */dev/nd0,1,...* ; but the server needs to reference these client partitions from time to time, to do *mkfs*(8) and *fsck*(8) for example. The */dev/ndl** entries allow the server 'local' access to his subpartitions without causing any net activity. The actual local unit number to client unit number correspondence is again recorded in the */etc/nd.local* text file.

The nd device driver is the same on both the client and server sides. There are no user level processes associated with either side, thus the latency and transfer rates are close to maximal.

The minor device and ioctl encoding used is given in file *<sun/ndio.h>*. The low six bits of the minor number are the unit number. The 0x40 bit indicates a *public* unit; the 0x80 bit indicates a *local* unit.

INITIALIZATION

No special initialization is required on the client side; he finds the server by broadcasting the initial request. Upon getting a response, he locks onto that server address.

At the server, the *nd*(8C) command initializes the network disk service by issuing ioctl's to the kernel.

ERRORS

Generally physical disk I/O errors detected at the server are returned to the client for action. If the server is down or unaccessable, the client will see the console message:

nd: file server not responding: still trying.

The client continues (forever) making his request until he gets positive acknowledgement from the server.

This means the server can crash or power down and come back up without any special action required of the user at the client machine. It also means the process performing the I/O to *nd* will block, insensitive to signals, since the process is sleeping inside the kernel at PRIBIO.

PROTOCOL AND DRIVER INTERNALS

The protocol packet is defined in *<sun/ndio.h>* and also included below:

```
/*
 * 'nd' protocol packet format.
 */
struct ndpack {
        struct  ip np_ip;/* ip header, proto IPPROTO_ND */
        u_char  np_op;          /* operation code, see below */
        u_char  np_min;         /* minor device */
        char    np_error;       /* b_error */
        char    np_ver;         /* version number */
        long    np_seq;         /* sequence number */
        long    np_blkno;       /* b_blkno, disk block number */
        long    np_bcount;      /* b_bcount, byte count */
        long    np_resid;       /* b_resid, residual byte count */
        long    np_caddr;       /* current byte offset of this packet */
        long    np_ccount;      /* current byte count of this packet */
};                      /* data follows */

/*
 * np_oe operation codes.
 */
#define NDOPREAD    1           /* read */
#define NDOPWRITE   2           /* write */
#define NDOPERROR   3           /* error */
#define NDOPCODE    7           /* op code mask */
#define NDOPWAIT    010         /* waiting for DONE or next request */
#define NDOPDONE    020         /* operation done */

/*
 * misc protocol defines.
 */
#define NDMAXDATA   1024        /* max data per packet  */
#define NDMAXIO     63*1024     /* max np_bcount */
```

IP datagrams were chosen instead of UDP datagrams because only the IP header is checksummed, not the entire packet as in UDP. Also the kernel level interface to the IP layer is simpler. The *min*, *blkno*, and *bcount* fields are copied directly from the client's strategy request. The sequence number field *seq* is incremented on each new client request and is matched with incoming server responses. The server essentially echos the request header in his responses, altering certain fields. The *caddr* and *ccount* fields show the current byte address and count of the data in this packet, or the data expected to be sent by the other side.

The protocol is very simple and driven entirely from the client side. As soon as the client ndstrategy routine is called, the request is sent to the server; this allows disk sorting to occur at the server as soon as possible. Transactions which send data (client writes on the client side, client reads on the server side) can only send a set number of packets of NDMAXDATA bytes each, before waiting for an acknowledgement. The defaults are currently set at 6 packets of 1K bytes each; the NDIOCETHER ioctl allows setting this value on the server side. This allows the normal 4K byte case to occur with just one 'transaction'. The NDOPWAIT bit is set in the *op* field by the sender to indicate he will send no more until acknowledged (or requested) by the other side. The NDOPDONE bit is set by the server side to indicate the request operation has completed; for both the read and write cases this means the requested disk I/O has actually occured.

Requests received by the server are entered on an active list which is timed out and discarded if not completed within NDXTIMER seconds. Requests received by the server allocate a *bcount* size buffer to minimize buffer copying. Contiguous DMA disk I/O thus occurs in the same size chunks it would if requested from a local physical disk.

BOOTSTRAP

The Sun workstation has PROM code to perform a net boot using this driver. Usually, the boot files are obtained from public device 0 (*/dev/ndp0*) on the server with which the client is registered; this allows multiple servers to exist on the same net (even running different releases of kernel and boot software). If the station you are booting is not registered on any of the servers, you will have to specify the hex Internet host number of the server in a boot command string like: 'bec(0,5,0)vmunix'.

This booting performs exactly the same steps involved in a real disk boot:

1) User types 'b' to PROM monitor.
2) PROM loads blocks 1 thru 15 of */dev/ndp0* (*bootnd*).
3) *bootnd* loads */boot*.
4) */boot* loads */vmunix*.

SEE ALSO

ioctl(2), nd(8C)

BUGS

The operations described in *dkio*(4) are not supported.

The local host's disk buffer cache is not used by network disk access. This means that if either a local host or a remote host is writing, the changes will be visible at random based on the cache hit frequency on the local host. Use *sync* on the server to force the data out to disk. If both the local and remote hosts are writing to the same filesystem, one machine's changes can be randomly lost, based again on cache hit and deferred write timings.

If an R/O remote file system is mounted R/W by mistake, it is impossible to umount it.

NAME

nfs, NFS – network file system

SYNOPSIS

options NFS

DESCRIPTION

The Network File System, or NFS, allows a client workstation to perform transparent file access over the network. Using it, a client workstation can operate on files that reside on a variety of servers, server architectures and across a variety of operating systems. Client file access calls are converted to NFS protocol requests, and are sent to the server system over the network. The server receives the request, performs the actual file system operation, and sends a response back to the client.

The Nework File System operates in a stateless fashion using remote procedure (RPC) calls built on top of external data representation (XDR) protocol. These protocols are documented in *Networking on the Sun Workstation*. The RPC protocol provides for version and authentication parameters to be exchanged for security over the network.

A server can grant access to a specific filesystem to certain clients by adding an entry for that filesystem to the server's */etc/exports* file.

A client gains access to that filesystem with the *mount*(2) system call, which requests a file handle for the filesystem itself. Once the filesystem is mounted by the client, the server issues a file handle to the client for each file (or directory) the client accesses. If the file is somehow removed on the server side, the file handle becomes stale (dissociated with a known file).

A server may also be a client with respect to filesystems it has mounted over the network, but its clients cannot gain access to those filesystems. Instead, the client must mount a filesystem directly from the server on which it resides.

The user ID and group ID mappings must be the same between client and server. However, the server maps uid 0 (the super-user) to uid –2 before performing access checks for a client. This inhibits super-user privileges on remote filesystems.

NFS-related routines and structure definitions are described in the *NFS Protocol Spec.* in *Networking on the Sun Workstation.*

ERRORS

Generally physical disk I/O errors detected at the server are returned to the client for action. If the server is down or inaccessable, the client will see the console message:

NFS: file server not responding: still trying.

The client continues (forever) to resend the request until it receives an acknowledgement from the server. This means the server can crash or power down, and come back up, without any special action required by the client. It also means the client process requesting the I/O will block and remain insensitive to signals, sleeping inside the kernel at PRIBIO.

SEE ALSO

exports(5), fstab(5), mntent(5), mount (2), mount(8), nfsd(8)

NAME

nit – Network Interface Tap Protocol

SYNOPSIS

options NIT

DESCRIPTION

nit is a provisional protocol familty which runs on top of the kernel raw socket code and provides the superuser with a tee connection into a specified network interface. For example, it provides the unprocessed packet read and write capability on the Ethernet interface *ie*(4S).

nit uses two structures to communicate information, the *nit_ioc* structure, which contains the ioctl information used to set parameter values; and the *nit_hdr* structure, which contains per packet statistics and is prepended to every delivered packet. When setting parameters, values that are otherwise impossible mean "don't change".

nit collects incoming packets into chunks to reduce the per packet overhead. The chunks are returned by *read*(2V) and *recv*(2) system calls. Outgoing packets are not buffered. The ioctl value *nioc_chunksize* sets the size of the incoming chunk. *Nioc_bufalign* and *nioc_bufoffset* control packet placement within buffers. The (nit) header for each packet in a buffer starts *nioc_bufoffset* bytes past some multiple of *nioc_bufalign* bytes from the beginning. The packet itself appears immediately beyond the header. *nit* also limits the amount of buffer space consumed. To change the default, set *nioc_bufspace*.

nit performs packet filtering and data selection on incoming packets. The data selection criterion is the length of the initial portion of the data packet to return to the user. The filtering criteria are packet destination and packet type. The filtering and data selection criteria are set via *nioc_snaplen*, *nioc_flags*, and *nioc_typetomatch*. The choices for destination are either normal or promiscuous. Normal destination filtering considers only those packets that are normally received by the machine running *nit* (both host specific and broadcast packets). Promiscuous destination filtering considers every packet visible on the network; this can place a large demand on the processor if there are many packets to receive. The packets are further filtered on type, an interface specific quantity. For the Ethernet interfaces, the type field is the packet type from the Ethernet header. See *<netinet/if_ether.h>*.

Outgoing packets are not (yet) handled in a general way, since there is no one address family which says "send the packet as is", where the data portion of the packet contains a complete packet to be transmitted without further processing. Therefore, in general, you can't send arbitrary packets. For the Ethernet, however, the address family AF_UNSPEC is defined so that the remaining 14 bytes of the *sockaddr* correspond the the first 14 bytes of the outgoing packet, which are the (6 byte) destination address, the (6 byte) source address (possibly overridden), and the (2 byte) type. See struct *ether_header* in *<netinet/if_ether.c>*. Therefore, for Ethernet in particular, it is possible to transmit an arbitrary packet. In the example which follows, *rarp_write* accepts an arbitrary packet and performs the interface specific manipulations required to transmit that packet.

The following definitions are taken from *<net/nit.h>*.

```
#define NITIFSIZ  10          /* size of ifname in sockaddr */
#define NITBUFSIZ 1024        /* buffers are rounded up to a
                               * multiple of this size (MCLBYTES) */

struct sockaddr_nit {
      u_short     snit_family;
      caddr_t     snit_cookie;            /* link to filtering */
      char  snit_ifname[NITIFSIZ];        /* interface name (eg, ie0) */
};

/* Header preceeding each packet returned to user */
struct nit_hdr {
      int   nh_state;         /* state of tap -- see below */
      struct timeval nh_timestamp;  /* time of arriving packet */
      int   nh_wirelen;       /* length (with header) off wire */
```

```
        union {
                int     info;           /* generic information */
                int     datalen;        /* length of saved packet portion */
                int     dropped;        /* number of dropped matched packets */
                int     seqno;          /* sequence number */
        } nh_un;
};
#define nh_info         nh_un.info
#define nh_datalen      nh_un.datalen
#define nh_dropped      nh_un.dropped
#define nh_seqno        nh_un.seqno

/* Ioctl parameter block */
struct nit_ioc {
        int     nioc_bufspace;          /* total buffer space to use */
        int     nioc_chunksize;         /* size of chunks to send */
        u_int nioc_typetomatch;         /* magic type with which to match */
        int     nioc_snaplen;           /* length of packet portion to snap */
        int     nioc_bufalign;          /* packet header alignment multiple */
        int     nioc_bufoffset;         /* packet header alignment offset */
        struct  timeval nioc_timeout;   /* delay after packet before drain */
        int     nioc_flags;             /* see below */
};
#define NT_NOTYPES      ((u_int)0)  /* match no packet types */
#define NT_ALLTYPES     ((u_int)-1) /* match all packet types */

#define NF_PROMISC      0x01        /* enter promiscuous mode */
#define NF_TIMEOUT      0x02        /* timeout value valid */
#define NF_BUSY         0x04        /* buffer is busy (has data) */

/*
 * States for the packet capture portion of nit,
 * some of which are passed to the user.
 */
#define NIT_QUIET       0       /* inactive */
#define NIT_CATCH       1       /* capturing packets */
#define NIT_NOMBUF      2       /* discarding -- out of mbufs */
#define NIT_NOCLUSTER   3       /* discarding -- out of mclusters */
#define NIT_NOSPACE     4       /* discarding -- would exceed bufspace */
/* Pseudo-states returned in information packets */
#define NIT_SEQNO       5       /* sequence number of chunk */
```

To use *nit*:

- Include definitions and declare needed variables, for example

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/nit.h>
#include <net/if.h>

struct sockaddr_nit snit;
struct nit_ioc nioc;
```

- Create a socket with the call
```
s = socket(AF_NIT, SOCK_RAW, NITPROTO_RAW);
```
- Bind it to an interface with a code fragment like
```
snit.snit_family = AF_NIT;
strncpy(snit.snit_ifname, "ie0", sizeof (snit.snit_ifname));
bind(s, (struct sockaddr *)&snit, sizeof (snit));
```
- To establish the operating modes, issue an ioctl; for example
```
bzero(&nioc, sizeof(nioc));
nioc.nioc_bufspace = NITBUFSIZ;
nioc.nioc_chunksize = NITBUFSIZ;
nioc.nioc_typetomatch = NT_ALLTYPES;
nioc.nioc_snaplen = 32767;
nioc.nioc_flags = NF_TIMEOUT;
nioc.nioc_timeout.tv_usec = 200;
if (ioctl(if_fd, SIOCSNIT, &nioc) != 0) {
      perror("nit ioctl");
      exit(2);
}
```
- To receive packets, issue *read*s (or *recv*s). To transmit packets, issue *write*s (or *send*s). For example, the following routine will transmit an arbitrary packet (including address information) on the Ethernet. Note that the Ethernet addresses and type are provided in the incoming buffer *buf*, and must be moved into the *sockaddr* destination address to satisfy the kernel.
```
rarp_write(fd, buf, len)
      int fd, len;
      char *buf;
{
      struct sockaddr sa;
      int offset = sizeof(sa.sa_data);
      int result;

      sa.sa_family = AF_UNSPEC;
      bcopy(buf, sa.sa_data, offset);
      result = sendto(fd, buf+offset, len-offset,
          0, &sa, sizeof(sa));
      return (result+offset);
}
```

SEE ALSO

bind(2), config(8), ec(4S), ie(4S), if(4N), ioctl(2), read(2V), recv(2), send(2), socket(2), write(2V).

Network Implementation in *Networking on the Sun Workstation*.

BUGS

This protocol is provisional, and is subject to change.

Buffering is limited to 32767 bytes.

Interface ioctl's may have different semantics on a nit socket.

nit is unable to see outgoing transmissions on some interfaces.

The selection criteria is very simplistic. Therefore, many packets may be passed to the user program, especially in promiscuous mode.

NAME

null – data sink

SYNOPSIS

None; included with standard system.

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return an end-of-file indication.

FILES

/dev/null

NAME

pty – pseudo terminal driver

SYNOPSIS

pseudo-device pty

DESCRIPTION

The *pty* driver provides support for a pair of devices collectively known as a *pseudo-terminal*. The two devices comprising a pseudo-terminal are known as a *master* and a *slave*. The slave device provides an interface identical to that described in *tty*(4), but instead of having a hardware interface such as the Zilog chip and associated hardware used by *zs*(4S) supporting the terminal functions, the functions of the terminal are implemented by another process manipulating the master side of the pseudo-terminal.

The master and the slave sides of the pseudo-terminal are tightly connected. Any data written on the master device is given to the slave device as input, as though it had been received from a hardware interface. Any data written on the slave terminal can be read from the master device (rather than being transmitted from a UART).

In configuring, if no optional "count" is given in the specification, 16 pseudo terminal pairs are configured.

A few special ioctl's are provided on the control-side devices of pseudo-terminals to provide the functionality needed by applications programs to emulate real hardware interfaces:

TIOCSTOP
: Stops output to a terminal (that is, like typing ^S). Takes no parameter.

TIOCSTART
: Restarts output (stopped by TIOCSTOP or by typing ^Q). Takes no parameter.

There are also two independent modes which can be used by applications programs:

TIOCPKT
: Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

  TIOCPKT_FLUSHREAD
  : whenever the read queue for the terminal is flushed.

  TIOCPKT_FLUSHWRITE
  : whenever the write queue for the terminal is flushed.

  TIOCPKT_STOP
  : whenever output to the terminal is stopped a la ^S.

  TIOCPKT_START
  : whenever output to the terminal is restarted.

  TIOCPKT_DOSTOP
  : whenever *t_stopc* is ^S and *t_startc* is ^Q.

  TIOCPKT_NOSTOP
  : whenever the start and stop characters are not ^S/^Q.

  This mode is used by *rlogin*(1C) and *rlogind*(8C) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output when interrupts occur; it can be used by other similar programs.

TIOCREMOTE
: A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes

input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal; a write of 0 bytes is like typing an end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required.

FILES

/dev/pty[p-r][0-9a-f]	master pseudo terminals
/dev/tty[p-r][0-9a-f]	slave pseudo terminals

BUGS

It is apparently not possible to send an EOT by writing zero bytes in TIOCREMOTE mode.

NAME

routing – system supporting for local network packet routing

DESCRIPTION

The network facilities provided general packet routing, leaving routing table maintenance to applications processes.

A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*:

```
struct rtentry {
        u_long  rt_hash;
        struct  sockaddr rt_dst;
        struct  sockaddr rt_gateway;
        short   rt_flags;
        short   rt_refcnt;
        u_long  rt_use;
        struct  ifnet *rt_ifp;
};
```

with *rt_flags* defined from:

```
#define RTF_UP          0x1     /* route usable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through it is a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface may be requested to address the packet to an entity different from the eventual recipient (i.e. the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the resources associated with it will not be reclaimed until all references to it are removed.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existant entry, or ENOBUFS if insufficient resources were available to install a new route.

User processes read the routing tables through the */dev/kmem* device.

The *rt_use* field contains the number of packets sent along the route. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

SEE ALSO

route(8C), routed(8C)

NAME

sd – Disk driver for SCSI Disk Controllers

SYNOPSIS — SUN-3

```
controller sc0 at vme24d16 ? csr 0x200000 priority 2 vector scintr 0x40
controller si0 at vme24d16 ? csr 0x200000 priority 2 vector siintr 0x40
controller si0 at obio ? csr 0x140000 priority 2
disk sd0 at sc0 drive 0 flags 0
disk sd1 at sc0 drive 1 flags 0
disk sd0 at si0 drive 0 flags 0
disk sd1 at si0 drive 1 flags 0
disk sd2 at sc0 drive 8 flags 0
disk sd2 at si0 drive 8 flags 0
```

The first two **controller** lines above specify the first SCSI host adapter on a Sun-3/160. The third **controller** line above specifies the first and only SCSI host adapter on a Sun-3/50. The first four **disk** lines specify the first and second disk drives on the first SCSI controller in a system. The last two **disk** lines specify the first disk drive on the second SCSI controller in a system.

The **drive** value is calculated using the formula:

$$8 * target + unit$$

where *target* is the SCSI target (controller number on host adapter), and *unit* is the SCSI logical unit.

SYNOPSIS — SUN-2

```
controller sc0 at mbmem ? csr 0x80000 priority 2
controller sc1 at mbmem ? csr 0x84000 priority 2
controller sc0 at vme24 ? csr 0x200000 priority 2 vector scintr 0x40
disk sd0 at sc0 drive 0 flags 0
disk sd1 at sc0 drive 1 flags 0
disk sd2 at sc1 drive 0 flags 0
disk sd3 at sc1 drive 1 flags 0
```

The first two **controller** lines above specify the first and second SCSI host adapters on a Sun-2/120 or Sun-2/170. The third **controller** line above specifies the first host adapter on a Sun-2/160. The four **disk** lines specify the first and second disk drives on the first and second SCSI controllers in a system (where each SCSI controller is on a different host adapter).

The **drive** value is calculated as described above.

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0. The standard device names begin with "sd" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block-files access the disk using the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface that provides for direct transmission between the disk and the user's read or write buffer. A single read or write call usually results in one I/O operation; therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O, requests to the SCSI disk must have an offset on a 512 byte boundary, and their length must be a multiple of 512 bytes or the driver will return an error (EINVAL). Likewise *seek* calls should specify a multiple of 512 bytes.

DISK SUPPORT

This driver handles all ST-506 and ESDI drives (assuming the correct controller is installed), by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The sd?a partition is normally used for the root file system on a disk, the sd?b partition as a paging area, and the sd?c partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the sd?g partition.

FILES

/dev/sd[0-7][a-h] block files
/dev/rsd[0-7][a-h] raw files

SEE ALSO

dkio(4S)
Adaptec ACB 4000 and 5000 Series Disk Controllers OEM Manual
Emulex MD21 SCSI Disk Controller Programmer Reference Manual

DIAGNOSTICS

sd%d%c: *cmd how (msg)* **starting blk %d, blk %d (abs blk %d).**

A command such as read or write encountered a error condition (how): either it *failed*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready" or "sector not found". The *starting blk* is the first sector of the erroneous command, relative to the beginning of the partition involved. The *blk* is the sector in error, again relative to the beginning of the partition involved. The *abs blk* is the absolute block number of the sector in error.

NAME

st – Driver for Sysgen SC 4000 (Archive) and the Emulex MT-02 Tape Controller

SYNOPSIS — SUN-3

controller sc0 at vme24d16 ? csr 0x200000 priority 2 vector scintr 0x40
controller si0 at vme24d16 ? csr 0x200000 priority 2 vector siintr 0x40
controller si0 at obio ? csr 0x140000 priority 2
tape st0 at sc0 drive 32 flags 1
tape st0 at si0 drive 32 flags 1
tape st1 at sc0 drive 40 flags 1
tape st1 at si0 drive 40 flags 1

The first two **controller** lines above specify the first SCSI controller on a Sun-3/160. The third **controller** line above specifies the first and only SCSI controller on a Sun-3/50. The four **tape** lines specify the first and second tape drives on the first SCSI controller in a system.

The **drive** value is calculated using the formula:

8 * *target* + *unit*

where *target* is the SCSI target, and *unit* is the SCSI logical unit.

SYNOPSIS — SUN-2

controller sc0 at mbmem ? csr 0x80000 priority 2
controller sc0 at vme24 ? csr 0x200000 priority 2 vector scintr 0x40
controller sc1 at mbmem ? csr 0x84000 priority 2
tape st0 at sc0 drive 32 flags 1
tape st0 at sc1 drive 32 flags 1
tape st1 at sc0 drive 40 flags 1
tape st1 at sc1 drive 40 flags 1

The first two **controller** lines above specify the first and second SCSI controllers on a Sun-2/120 or Sun-2/170. The third **controller** line specifies the first controller on a Sun-2/160. The four **tape** lines specify the first and second tape drives on the first and second SCSI controllers in a system.

The **drive** value is calculated as described above.

DESCRIPTION

The Sysgen tape controller is a SCSI bus interface to an Archive streaming tape drive. It provides a standard tape interface to the device, see *mtio*(4), with some deficiencies listed under BUGS below. To utilize the QIC 24 format, access the logical device that is eight more than the default physical (QIC 11) device (that is, rst0 = QIC 11, rst8 = QIC 24).

FILES

/dev/rst[0-3]	QIC 11 Format
/dev/rst[8-11]	QIC 24 Format
/dev/nrst[0-3]	non-rewinding QIC 11 Format
/dev/nrst[8-11]	non-rewinding QIC 24 Format

SEE ALSO

mtio(4)

Sysgen SC4000 Intelligent Tape Controller Product Specification

DIAGNOSTICS

st*: tape not online.
st*: no cartridge in drive.
st*: cartridge is write protected.
st*: format change failed.
st*: device not supported.

BUGS

The tape cannot reverse direction so the BSF and BSR ioctls are not supported.

The FSR ioctl is not supported.

Most disk I/O over the SCSI bus is prevented when the tape is in use. This is because the controller does not free the bus while the tape is in motion (even during rewind).

When using the raw device, the number of bytes in any given transfer must be a multiple of 512. If it is not, the device driver returns an error.

The driver will only write an end of file mark on close if the last operation was a write, without regard for the mode used when opening the file. This will cause empty files to be deleted on a raw tape copy operation.

Some older systems may not support the QIC 24 device, and may complain (or exhibit erratic behavior) when the user attempts a QIC 24 device access.

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

None; included automatically with *inet*(4F).

DESCRIPTION

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit switched networks.

TCP fits into a layered protocol architecture just above the basic Internet Protocol (IP) described in *ip*(4P) which provides a way for TCP to send and receive variable-length segments of information enclosed in Internet datagram "envelopes." The Internet datagram provides a means for addressing source and destination TCPs in different networks, deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple netwokrs and interconnecting gateways, and has the ability to carry information on the precedence, security classification and compartmentalization of the TCP segments (although this is not currently implemented under the UNIX system.)

An application process interfaces to TCP through the *socket*(2) abstraction and the related calles *bind*(2), *listen*(2), *accept*(2), *connect*(2), *send*(2) and *recv*(2). The primary purpose of TCP is to provide a reliable bidirectional virtual circuit service between pairs of processes. In general, the TCP's decide when to block and forward data at their own convenience. In the UNIX system implementation, it is assumed that any buffering of data is done at the user level, and the TCP's transmit available data as soon as possible to their remote peer. They do this and always set the PUSH bit indicating that the transferred data should be made available to the user process at the remote end as soon as practicable.

To provide reliable data TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the underlying internet communications system. This is achieved by assigning a sequence number to each byte of data transmitted and requiring a positive acknowledgement from the receiving TCP. If the ACK is not received within an (adaptively determined) timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments. As long as the TCP's continue to function properly and the internet system does not become disjoint, no tranmission errors will affect the correct delivery of data, as TCP recovers from communications errors.

TCP provides flow control over the transmitted data. The receiving TCP is allowed to specify the amount of data which may be sent by the sender, by returning a *window* with every acknowledgement indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of bytes that the sender may transmit before receiving further permission.

TCP extends the standard 32-bit Internet host addresses with a 16-bit port number space; the combined addresses are available at the UNIX system process level in the standard *sockaddr_in* format described in *inet*(4F).

Sockets utilizing the tcp protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket to an address with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address

INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network. See *inet*(4F) for a complete description of addressing in the Internet family.

A TCP connection is created at the server end by doing a *socket*(2), a *bind*(2) to establish the address of the socket, a *listen*(2) to cause connection queueing, and then an *accept*(2) which returns the descriptor for the socket. A client connects to the server by doing a *socket*(2) and then a *connect*(2). Data may then be sent from server to client and back using *read*(2V) and *write*(2V).

TCP implements a very weak out-of-band mechanism, which may be invoked using the out-of-band provisions of *send*(2). This mechanism allows setting an urgent pointer in the data stream; it is reflected to the TCP user by making the byte after the urgent pointer available as out-of-band data and providing a SIOCATMARK ioctl which returns an integer indicating whether the stream is at the urgent mark. The system never returns data across the urgent mark in a single read. Thus, when a SIGURG signal is received indicating the presence of out-of-band data, and the out-of-band data indicates that the data to the mark should be flushed (as in remote terminal processing), it suffices to loop, checking whether you are at the out-of-band mark, and reading data while you are not at the mark.

SEE ALSO

inet(4F), ip(4P)

BUGS

It should be possible to send and receive TCP options.

The system always tries to negotiates the maximum TCP segment size to be 1024 bytes. This can result in poor performance if an intervening network performs excessive fragmentation.

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

NAME

termio – general terminal interface

SYNOPSIS

None; included by default.

DESCRIPTION

This section describes the special file **/dev/tty** and the terminal drivers used for interactive I/O by devices such as *zs*(4S), *cons*(4S), and *pty*(4).

Opening a Terminal File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *init (8)* and become a user's standard input, output, and error files.

The Controlling Terminal

A terminal may belong to a process, in which case it is known as its *controlling terminal*. This controlling terminal may have a distinguished process group associated with it which plays a special role in handling QUIT and INT (interrupt) signals, as discussed below. The controlling terminal is inherited by a child process during a *fork*(2).

If a process that has no controlling terminal opens a terminal file, then the device or pseudo-device associated with that terminal file becomes the controlling terminal for the process; the terminal's distinguished process group is set to that of the process.

The file **/dev/tty** is, for each process, a synonym for its controlling terminal. This is useful for programs that wish to be sure of writing messages on the terminal directly, no matter how output has been redirected. It can also be used for programs that demand a filename for output when typed output is desired and it is tiresome to find out which terminal is currently in use.

Implementation Restrictions

Due to restrictions imposed by the current terminal driver, some features are not fully supported:

1. Certain terminal driver features are always enabled, except when the driver is in "RAW mode". If the character size is 8 bits, no parity is specified, no output processing is selected (i.e., either OPOST is on or none of OLCUC, ONLCR, or any of the delays are selected), and no input process is selected (as with BRKINT, IGNPAR, INPCK, ISTRIP, ICRNL, IUCLC, and IXON are all off in the *c_iflag* word and ISIG, ICANON, and XCASE are all off in the *c_lflag* word), the driver is in "RAW mode". If a terminal port is being used for transferring binary data (such as when *uucp*(1) or some microcomputer data transfer program like KERMIT is using the port), it is usually in "RAW mode". If it is being used to give a user interactive access to the computer, it is usually not in "RAW mode".

   BRKINT and IGNPAR are disabled only in "RAW mode"; if they are to be disabled, the other modes listed must also be disabled. The WERASE, REPRINT, DISCARD, and LNEXT characters are also disabled only in raw mode.

2. IUCLC, OLCUC, and XCASE must either all be on or all be off, and ICRNL and ONLCR must either both be on or both be off.

3. The MIN and TIME values supported by other implementations can be set, but this has no effect on the terminal driver. The driver behaves as if MIN were 1 and TIME were 0.

4. Character sizes CS5 and CS6 may not be selected; size CS7 may only be selected when PARENB is set, and size CS8 may only be selected when PARENB is not set. Furthermore, if size CS8 is selected, unless OPOST is not set, it only applies to input, not output.

5. IGNBRK, PARMRK, INLCR, IGNCR, ONOCR, OFILL, OFDEL and ECHONL are treated as if they were always not set. CREAD and ECHOK are treated as if they were always set.

6. The TCSETAW call will flush any pending input, just as TCSETAF does. TCSBRK will also flush any pending input.

7. The EOF character may not be escaped with a backslash (\) (unless XCASE is set; see below).

8. If XCASE is set, a backslash (\) followed by any character other than a letter or one of the special characters listed in the description of XCASE will be read as the character; the backslash will not be read. This includes the special characters ERASE, WERASE, KILL, REPRINT, EOF, NEWLINE (the ASCII NL character), EOL, and DISCARD.

Reading Characters

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character-input buffers become completely full (which is rare), or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, if the terminal port is in "RAW mode", all the saved characters are thrown away without notice. Otherwise, any further input is discarded and an ASCII BEL character is echoed.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode (see ICANON in the *Local Modes* section).

Canonical Mode Input Processing

In canonical mode, terminal input is processed in units of lines. A line is delimited by a NL (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters can be requested in a read, even one, without losing information.

Erase and kill processing is normally done during input. The ERASE character (by default, the character DEL) erases the last character typed. The WERASE character (the character ^W) erases the last "word" typed (but not any preceding spaces or tabs). A "word" is defined as a sequence of nonblank characters, with tabs counted as blanks. Neither ERASE nor WERASE will erase beyond the beginning of the line. The KILL character (by default, the character ^U) kills (deletes) the entire input line, and optionally produces a NL character. These special characters operate on a keystroke basis, independently of any backspacing or tabbing that may have been done.

The REPRINT character (the character ^R) prints a NL followed by all characters which have not been read. Reprinting also occurs automatically if characters which would normally be erased from the screen are fouled by program output. The characters are reprinted as if they were being echoed; as a consequence, if ECHO is not set, they are not printed.

The ERASE and KILL characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The ERASE and KILL characters may be changed by such commands as *stty*(1), and *tset*(1).

Noncanonical Mode Input Processing

In non-canonical mode, input characters are not assembled into lines, and erase and kill processing does not occur. Characters are read as soon as they are typed.

Writing Characters

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed as they are typed, if echoing has been enabled. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Special Characters

Certain characters have special functions on input and/or output. These functions and their default character values are summarized as follows:

INTR (Control-C or ASCII ETX) generates a SIGINT signal which is sent to all processes in the distinguished process group associated with the terminal. Normally, each such process is forced

to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *sigvec*(2).

QUIT (Control-| or ASCII FS) generates a SIGQUIT signal which is sent to all processes in the distinguished process group associated with the terminal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (Rubout or ASCII DEL) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

WERASE (^**W** or ASCII ETB) erases the preceding "word". It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (^**U** or ASCII NAK) deletes the entire line, as delimited by a NL, EOF, or EOL character.

REPRINT (^**R** or ASCII DC2) reprints all characters which have not been read, preceded by a NL.

EOF (^**D** or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a NL, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (Off by default) is an additional line delimiter, like NL. It is not normally used.

SUSP (^**Z** or ASCII EM) is used by the job control facility to change the current job to return to the controlling job. It generates a SIGTSTP signal, which stops all processes in the terminal's process group.

DSUSP (^**Y** or ASCII SUB) is used by the job control facility to change the current job to return to the controlling job. It generates a SIGTSTP signal as SUSP does, but the signal is sent when a program attempts to read the DSUSP character, rather than when it is typed.

STOP (^**S** or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (^**Q** or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

DISCARD (^**O** or ASCII SI) causes subsequent output to be discarded until another DISCARD character is typed, more input arrives, or the condition is cleared by a program.

LNEXT (^**V** or ASCII SYN) causes the special meaning of the next character to be ignored; this works for all the special characters mentioned above. This allows characters to be input that would otherwise get interpreted by the system (such as KILL, or QUIT).

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE and KILL characters may be escaped by a preceding \ character, in which case no special function is done. Any of the special characters may be preceded by the LNEXT character, in which case no special function is done.

When in "RAW mode", none of the special characters perform any special function.

Modem Disconnect

When the carrier signal from the data-set drops, a SIGHUP signal is sent to all processes in the distinguished process group associated with this terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If SIGHUP is ignored or caught, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate

appropriately when hung up on.

ioctl **Calls**

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define NCC        8
struct  termio {
        unsigned   short   c_iflag;     /* input modes */
        unsigned   short   c_oflag;     /* output modes */
        unsigned   short   c_cflag;     /* control modes */
        unsigned   short   c_lflag;     /* local modes */
        char               c_line;      /* line discipline */
        unsigned   char    c_cc[NCC];   /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	VINTR	ETX
1	VQUIT	FS
2	VERASE	DEL
3	VKILL	NAK
4	VEOF	EOT
5	VEOL	(disabled)
6	reserved	
7	reserved	

Input Modes

The *c_iflag* field describes the basic terminal input control:

BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

If BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored. A framing or parity error which is not ignored is read as the ASCII NUL character (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all eight bits are processed.

If ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character. Note: if this bit is set, the OLCUC bit in the *c_oflag* word and the XCASE bit in the *c_lflag* word must also be set.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are not read, but merely perform flow control functions. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit a STOP character when the input queue is nearly full, and a START character when enough input has been read that the input queue is nearly empty again.

The initial input control value is undefined.

Output Modes

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
ONLRET	0000040	NL performs CR function.
NLDLY	0000400	Select
NL		
delays:		
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays
TAB0	0	or tab expansion:
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. Note: if this bit is set, the IUCLC bit in the *c_iflag* word and the XCASE bit in the *c_lflag* word must also be set.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

NEWLINE
delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the NL delays.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.08 seconds, and type 3 is about 0.16 seconds.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is not supported. Type 3 specifies that tabs are to be expanded into spaces.

Backspace delay is not supported.

The actual delays depend on line speed and system load.

The initial output control value is undefined.

Control Modes

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	19200 baud
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stop bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used. The only combinations that are supported are CS7 with PARENB and CS8 without PARENB.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

The initial hardware control value after open is undefined.

Local Modes

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR, QUIT, SUSP, and DSUSP. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will be satisfied as soon as one character is received; the values of MIN and TIME are ignored.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\!
~	\^
{	\(
}	\)

Any other character, when preceded on input by \, will be read as itself, and the \ will not be read. This means a \ must be entered as \\. For example, A is input as \a, \n as \\n, and \N as \\\n. Note: if this bit is set, the IUCLC bit in the *c_iflag* word and the OLCUC bit in the *c_oflag* word must also be set.

If ECHO is set, characters are echoed as received. If ECHO is not set, input characters are not echoed.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as a sequence of ASCII BS SP BS, which will clear the last character from a CRT screen. If the baud rate is greater than 1200 baud, the kill character is echoed as a sequence of ASCII BS SP BS, which will clear all the characters on the current line from a CRT screen; otherwise, it is echoed as itself (if it is a control character, it will be echoed as described below) followed by an NL character. If ECHOE is not set, the erase character is echoed by printing the character being erased; erased characters are echoed between a backslash (\) and a slash (/). The NL character is always echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function.

Non-printing (control) characters are normally echoed as ^*X*, where *X* is the character given by adding 100 octal to the control character's code (so that the character with octal code 1 is echoed as ^A), and the ASCII DEL character, with code 177 octal, is echoed as ^?. In "RAW mode", control characters and DEL are echoed as themselves.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP characters will not be done.

The initial line-discipline control value is undefined.

The primary *ioctl*(2) system calls have the form:

ioctl (*fildes*, *command*, *arg*)
struct termio **arg*;

The commands using this form are:

TCGETA Get the parameters associated with the terminal and store in the *termio* structure referenced by *arg*.

TCSETA Set the parameters associated with the terminal from the structure referred to by *arg*. The change is immediate.

TCSETAW

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters. This form should be used when changing parameters that will affect output.

Additional *ioctl*(2) calls have the form:

ioctl (*fildes*, *command*, *arg*)
int *arg*;

The commands using this form are:

TCSBRK Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).

Note: this call will flush any pending input.

TCXONC Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.

TCFLSH If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

FILES

/dev/tty*

SEE ALSO

stty(1V), tset(1), fork(2), ioctl(2), setpgrp(2V), signal(2).

NAME

tm – tapemaster 1/2 inch tape drive

SYNOPSIS — SUN-3

controller tm0 at vme16d16 ? csr 0xa0 priority 3 vector tmintr 0x60
controller tm1 at vme16d16 ? csr 0xa2 priority 3 vector tmintr 0x61
tape mt0 at tm0 drive 0 flags 1
tape mt0 at tm1 drive 0 flags 1

SYNOPSIS — SUN-2

controller tm0 at mbio ? csr 0xa0 priority 3
controller tm0 at vme16 ? csr 0xa0 priority 3 vector tmintr 0x60
controller tm1 at mbio ? csr 0xa2 priority 3
controller tm1 at vme16 ? csr 0xa2 priority 3 vector tmintr 0x61
tape mt0 at tm0 drive 0 flags 1
tape mt0 at tm1 drive 0 flags 1

DESCRIPTION

The Tapemaster tape controller controls Pertec-interface 1/2" tape drives such as the CDC Keystone, providing a standard tape interface to the device, see *mtio*(4).

SEE ALSO

mt(1), tar(1), ar(4S)

DIAGNOSTICS

tm*n*: no response from ctlr.
tm*n*: error *n* during config.
mt*n*: not online.
mt*n*: no write ring.
tmgo: gate wasn't open. Controller lost synch.
tmintr: can't clear interrupts.
tm*n*: stray interrupts.
mt*n*: hard error bn=*n* er=%x.
mt*n*: lost interrupt.

BUGS

The Tapemaster controller does not provide for byte-swapping and the resultant system overhead prevents streaming transports from streaming.

If a non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

tty – general terminal interface

SYNOPSIS

None; included by default.

DESCRIPTION

This section describes the special file **/dev/tty** and the terminal drivers used for conversational computing by devices such as *zs*(4S), *cons*(4S), and *pty*(4).

Line disciplines.

The system provides different *line disciplines* for controlling communications lines. In this version of the system there are three disciplines available:

old The old (standard) terminal driver. This is used when using the standard shell *sh*(1) and for compatibility with Version 7 UNIX systems.

new A newer terminal driver, with features for job control; this must be used when using *csh*(1).

net A line discipline used for networking and loading data into the system over communications lines. It allows high speed input at very low overhead, and is described in *bk*(4).

Line discipline switching is accomplished with the TIOCSETD *ioctl*:

```
int ldisc = LDISC;
ioctl(f, TIOCSETD, &ldisc);
```

where LDISC is OTTYDISC for the standard tty driver, NTTYDISC for the new driver and NETLDISC for the networking discipline. The standard (currently old) tty line discipline is 0 by convention. The current line discipline can be obtained with the TIOCGETD *ioctl*. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved. The remainder of this section discusses the "old" and "new" disciplines.

Opening a terminal device file.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by *init*(8) and become a user's standard input, output, and error files.

The controlling terminal.

A terminal may belong to a process as its *controlling terminal*, and may have a distinguished process group associated with it. This distinguished process group plays a special role in handling quit and interrupt signals, as discussed below.

If a process which has no controlling terminal opens a terminal file, then the terminal associated with that terminal file becomes the controlling terminal for that process, and the terminal's distinguished process group is set to the process group of that process. The control terminal is thereafter inherited by a child process during a *fork*(2), even if the control terminal is closed.

The file **/dev/tty** is, in each process, a synonym for that process' controlling terminal. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

A process can remove the association it has with its controlling terminal by opening the file **/dev/tty** and issuing a

```
ioctl(f, TIOCNOTTY, 0);
```

This is often desirable in server processes.

Process groups.

Command processors such as *csh*(1) can arbitrate the terminal between different *jobs* by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the TIOCSPGRP *ioctl*(2):

ioctl(fildes, TIOCSPGRP, &pgrp);

or examined using TIOCGPGRP, which returns the current process group in *pgrp*. The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see **Job access control** below.

Modes.

The terminal line disciplines have three major modes, characterized by the amount of processing on the input and output characters:

cooked — The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when the *t_brkc* character, normally an EOT (control-D, hereafter ^D), is entered. A carriage return is usually made synonymous with newline in this mode, and replaced with a newline whenever it is typed. All line discipline functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

CBREAK — This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

RAW — This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when the terminal is put in non-blocking I/O mode; see the FNDELAY flag as described in *fcntl*(2). In this case a *read*(2V) from the control terminal will never block, but rather return an error indication (EWOULDBLOCK) if there is no input available.

A process may also request a SIGIO signal be sent it whenever input is present. To enable this mode the FASYNC flag should be set using *fcntl*(2).

Input editing.

A UNIX system terminal ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. In RAW mode, the terminal driver throws away all input and output without notice when the limit is reached. In CBREAK or cooked mode it refuses to accept any further input and, if in the new line discipline, rings the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word it is possible to have input characters with that parity discarded (see the **Summary** below.)

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI *ioctl*, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal except for the super-user (this call is not in standard Version 7 UNIX systems).

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the *stty*(3C) call or the TIOCSETN or TIOCSETP *ioctl*s (see the **Summary** below).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (but see the description of SIGTTIN in **Job access control** and of FIONREAD in **Summary**, both below.) No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, line editing is normally done, with the DELETE character logically erasing the last character typed and a ^U (control-U) logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or an ^D. These characters may be entered literally by preceding them with '\'; the '\' will normally be erased when the character is typed.

The line disciplines normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new line discipline there is a literal-next character ^V which can be typed in both cooked and CBREAK mode preceding *any* character to prevent its special meaning. This is to be preferred to the use of '\' escaping erase and kill characters, but '\' is retained with its old function in the new line discipline.

The new terminal line discipline also provides two other editing characters in normal mode. The word-erase character, normally ^W, erases the preceding word, but not any spaces before it. For the purposes of ^W, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally ^R, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode if characters which would normally be erased from the screen are fouled by program output.

Input echoing and redisplay

The terminal driver has several modes (not present in standard UNIX Version 7 systems) for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals. When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

CRT terminals. When a CRT terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal line discipline then echoes the proper number of backspace characters when input is erased to reposition the cursor. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing characters from a CRT. When a CRT terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters. If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for use on CRT terminals are speed dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKIL processing is painfully slow, so *stty*(1) normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater all of these bits are normally set. The *stty*(1) command summarizes these option settings and the use of the new terminal line discipline as "newcrt."

Output processing.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has

drained down to some threshold the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using RAW or CBREAK mode should be careful.

The terminal line disciplines provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces ^H, form feeds ^L, carriage returns ^M, tabs ^I and newlines ^J. The line disciplines will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns, and optionally convert newlines to carriage returns followed by newline. These functions are controlled by bits in the tty flags word; see **Summary** below.

The terminal line disciplines provide for mapping between upper and lower case on terminals lacking lower case, and for other special processing on deficient terminals.

Finally, in the new terminal line discipline, there is an output flush character, normally ^O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. An *ioctl* to flush the characters in the input or output queues, TIOCFLUSH, is also available.

Upper case terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. Upper case letters are preceded by a '\' when output. In addition, the following escape sequences can be generated on output and accepted on input:

for	`	|	~	{	}
use	\'	\!	\^	\(	\)

To deal with Hazeltine terminals, which do not understand that ~ has been made into an ASCII character, the LTILDE bit may be set in the local mode word; in this case the character ~ will be replaced with the character ` on output.

Flow control.

There are two characters (the stop character, normally ^S, and the start character, normally ^Q) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode the system produces a stop character (default ^S) when the input queue is in danger of overflowing, and a start character (default ^Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

Line control and breaks.

There are several *ioctl* calls available to control the state of the terminal line. The TIOCSBRK *ioctl* will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with *sleep*(3)) by TIOCCBRK. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The TIOCCDTR *ioctl* will clear the data terminal ready condition; it can be set again by TIOCSDTR.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further reads will fail, and programs that read a terminal and test for end-of-file on their input will terminate appropriately.

When using an ACU it is possible to ask that the phone line be hung up on the last close with the TIOCHPCL *ioctl*; this is normally done on the outgoing line.

Interrupt characters.

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent to the processes in the control group of the terminal, as if a TIOCGPGRP *ioctl* were done to get the process group and then a *killpg*(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*a la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed.

^C **t_intrc** (ETX) generates a SIGINT signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

^\ **t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

^Z **t_suspc** (EM) generates a SIGTSTP signal, which is used to suspend the current process group.

^Y **t_dsuspc** (SUB) generates a SIGTSTP signal as ^Z does, but the signal is sent when a program attempts to read the ^Y, rather than when it is typed.

Job access control.

When using the new terminal line discipline, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring SIGTTIN, has SIGTTIN blocked, or is in the middle of process creation using *vfork*(2)), the read will return −1 and set *errno* to EIO.

When using the new terminal line discipline with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are holding or ignoring SIGTTOU signals or which are in the middle of a *vfork*(2) are excepted and allowed to produce output.

Summary of modes.

Unfortunately, due to the evolution of the terminal drivers and line disciplines, there are 4 different structures which contain various portions of the driver and line discipline data. The first of these (**sgttyb**) contains that part of the information largely common between Version 6 and Version 7 UNIX systems. The second contains additional control characters added in Version 7. The third is a word of local state added in 4BSD, and the fourth is another structure of special characters added for the new line discipline. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty.

The basic *ioctl*s use the structure defined in **<sgtty.h>**:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        short   sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in **<sys/ttydev.h>**.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	19200 baud
EXTB	15	External B

Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are DELETE and ^U.)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0040000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0002000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input and generated on output
ODDP	0000100	Odd parity allowed on input and generated on output
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; output LF as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input and lower to upper on output
CBREAK	0000002	Return each character as soon as typed

TANDEM 0000001 Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the concept-100 and pads lines to be at least 9 characters at 9600 baud.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Input characters with the wrong parity, as determined by bits 200 and 100, are ignored in cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode all data in the input and output queues are discarded; this applies to both the new and the old line disciplines.

CRMOD causes input carriage returns to be turned into new-lines, and output and echoed new-lines to be output as a carriage return followed by a line feed.

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ and EOT are disabled.

TANDEM mode causes the system to produce a "stop" character (default ^S) whenever the input queue is in danger of overflowing, and a "start" character (default ^Q) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is really another computer which understands the conventions.

Note: The same "stop" and "start" characters are used for both directions of flow control; the *t_stopc* character is accepted on input as the character that stops output and is produced on output as the character to stop input, and the *t_startc* character is accepted on input as the character that restarts output and is produced on output as the character to restart input.

Basic ioctls

A large number of *ioctl*(2) calls apply to terminals. Some have the general form:

#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;

The applicable codes are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to *sgttyb* structure.

TIOCSETP Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes *arg* is ignored.

TIOCEXCL Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

With the following codes *arg* is a pointer to an **int**.

TIOCGETD *arg* is a pointer to an **int** into which is placed the current line discipline number.

TIOCSETD *arg* is a pointer to an **int** whose value becomes the current line discipline number.

TIOCFLUSH If the **int** pointed to by *arg* has a zero value, all characters waiting in input or output queues are flushed. Otherwise, the value of the **int** is treated as the logical OR of the FREAD and FWRITE defined in **<sys/file.h>**; if the FREAD bit is set, all characters waiting in input queues are flushed, and if the FWRITE bit is set, all characters waiting in output queues are flushed.

The remaining calls are not available in vanilla Version 7 UNIX systems. In cases where arguments are required, they are described; *arg* should otherwise be given as 0.

TIOCSTI the argument points to a character which the system pretends had been typed on the terminal.

TIOCSBRK the break bit is set in the terminal.

TIOCCBRK the break bit is cleared.

TIOCSDTR data terminal ready is set.

TIOCCDTR data terminal ready is cleared.

TIOCSTOP output is stopped as if the "stop" character had been typed.

TIOCSTART output is restarted as if the "start" character had been typed.

TIOCGPGRP *arg* is a pointer to an **int** into which is placed the process group ID of the process group for which this terminal is the control terminal.

TIOCSPGRP *arg* is a pointer to an **int** (typically a process ID); the process group whose process group ID is the value of this **int** becomes the process group for which this terminal is the control terminal.

TIOCOUTQ returns in the **int** pointed to by *arg* the number of characters queued up to be output to the terminal.

FIONREAD returns in the **int** pointed to by *arg* the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals.

Tchars

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces: The following structure is defined in **<sys/ioctl.h>**, which is automatically included by **<sgtty.h>**:

```
struct tchars {
        char    t_intrc;        /* interrupt */
        char    t_quitc;        /* quit */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output */
        char    t_eofc;         /* end-of-file */
        char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are ^C, ^\, ^Q, ^S, ^D, and '\377'. A character value of '\377' eliminates the effect of that character. The *t_brkc* character, by default '\377', acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical. The applicable *ioctl* calls are:

TIOCGETC Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

Local mode

The third structure associated with each terminal is a local mode word. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ~ to ` on output (for Hazeltine terminals)
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
	001000	Unimplemented.
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LPASS8	004000	Pass all 8 bits through on input, in any mode
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC systems
LNOFLSH	100000	Inhibit flushing of pending I/O when an interrupt character is typed.

The applicable *ioctl* functions are:

TIOCLBIS *arg* is a pointer to an **int** whose value is a mask containing the bits to be set in the local mode word.

TIOCLBIC *arg* is a pointer to an **int** whose value is a mask containing the bits to be cleared in the local mode word.

TIOCLSET *arg* is a pointer to an **int** whose value is stored in the local mode word.

TIOCLGET *arg* is a pointer to an **int** into which the current local mode word is placed.

Local special chars

The final structure associated with each terminal is the *ltchars* structure which defines control characters for the new line discipline. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dsuspc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasc;       /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The default values for these characters are ^Z, ^Y, ^R, ^O, ^W, and ^V. A value of '\377' disables the character.

The applicable *ioctl* functions are:

TIOCSLTC *arg* is a pointer to an *ltchars* structure which defines the new local special characters.

TIOCGLTC *arg* is a pointer to an *ltchars* structure into which is placed the current set of local special characters.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

csh(1), stty(1), ioctl(2), sigvec(2), stty(3C), getty(8), init(8)

BUGS

Half-duplex terminals are not supported.

Processes that are not invoked with a control terminal, but open a *dialout* line can hang indefinitely. Once the *dialout* line is opened, it becomes the control terminal. Should the process then open **/dev/tty**, it will hang because **/dev/tty** resolves to the corresponding *dialin* line. The process will wait for the dialin sequence to complete, even though the line is already connected.

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

None; comes automatically with *inet*(4F).

DESCRIPTION

The User Datagram Protocol (UDP) is defined to make available a datagram mode of packet switched computer communicaton in the environment of an interconnected set of computer networks. The protocol assumes that the Internet Protocol (IP) as described in *ip*(4P) is used as the underlying protocol.

The protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) as described in *tcp*(4P).

The UNIX system implementation of UDP makes it available as a socket of type SOCK_DGRAM. UDP sockets are normally used in a connectionless fashion, with the *sendto* and *recvfrom* calls described in *send*(2) and *recv*(2).

A UDP socket is created with a *socket*(2) call:

s = socket(AF_INET, SOCK_DGRAM, 0);

The socket initially has no address associated with it, and may be given an address with a *bind*(2) call as described in *inet*(4F). If no *bind* call is done, then the address assignment procedure described in *inet*(4F) is repeated as each datagram is sent.

When datagrams are sent the system encapsulates the user supplied data with UDP and IP headers. Unless the invoker is the super-user datagrams which would become broadcast packets on the network to which they are addressed are not allowed. Unless the socket has had a SO_DONTROUTE option enabled (see *socket*(2)) the outgoing datagram is routed through the routing tables as described in *routing*(4N). If there is insufficient system buffer space to temporarily hold the datagram while it is being trasmitted, the *sendto* may result in a ENOBUFS error. Other errors (ENETUNREACH, EADDRNOTAVAIL, EACCES, EMSGSIZE) may be generated by *icmp*(4P) or by the network interfaces themselves, and are reflected back in the *send* call.

As each UDP datagram arrives at a host the system strips out the IP options and checksums the data field, discarding the datagram if the checksum indicates that the datagram has been damaged. If no socket exists for the datagram to be sent to then an ICMP error is returned to the originating socket. If a socket exists for this datagram to be sent to, then we will append the datagram and the address from which it came to a queue associated with the datagram socket. This queue has limited capacity (2048 bytes of datagrams) and arriving datagrams which will not fit within its *high-water* capacity are silently discarded.

UDP processes ICMP errors reflected to it by *icmp*(4P). QUENCH errors are ignored (this is well considered a bug); UNREACH, TIMXCEED and PARAMPROB errors cause the socket to be disconnected from its peer if it was bound to a peer using *bind*(2) so that subsequent attempts to send datagrams via that socket will give an error indication.

The UDP datagram protocol differs from IP datagrams in that it adds a checksum over the data bytes and contains a 16-bit socket address on each machine rather than just the 32-bit machine address; UDP datagrams are addressed to sockets; IP packets are addressed to hosts.

SEE ALSO

recv(2), send(2), inet(4F)

"User Datagram Protocol," RFC768, John Postel, USC-ISI (Sun 800-1054-01)

BUGS

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

Something sensible should be done with QUENCH errors if the socket is bound to a peer socket.

NAME

vp – Ikon 10071-5 Versatec parallel printer interface

SYNOPSIS — SUN-2

device vp0 at mbio ? csr

DESCRIPTION

This Sun interface to the Versatec printer/plotter is supported by the Ikon parallel interface board, a word DMA device, which is output only.

The Versatec is normally handled by the line printer spooling system and should not be accessed by the user directly.

Opening the device */dev/vp0* may yield one of two errors: ENXIO indicates that the device is already in use; EIO indicates that the device is offline.

The printer operates in either print or plot mode. To set the printer into plot mode you should include <vcmd.h> and use the *ioctl*(2) call

```
ioctl(f, VSETSTATE, plotmd);
```

where *plotmd* is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

When going back into print mode from plot mode you normally eject paper by sending it an EOT after putting into print mode:

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(vp);
f = fileno (vp);
ioctl(f, VSETSTATE, prtmd);
write(f, "\04", 1);
```

FILES

/dev/vp0

BUGS

If you use the standard i/o library on the Versatec, be sure to explicitly set a buffer using *setbuf*, since the library will not use buffered output by default, and will run very slowly.

Writes must start on even byte boundaries and be an even number of bytes in length.

NAME

vpc – Systech VPC-2200 Versatec printer/plotter and Centronics printer interface

SYNOPSIS — SUN-2

device vpc0 at mbio ? csr 0x480 priority 2
device vpc1 at mbio ? csr 0x500 priority 2

DESCRIPTION

This Sun interface to the Versatec printer/plotter and to Centronics printers is supported by the Systech parallel interface board, an output-only byte-wide DMA device. The device has one channel for Versatec devices and one channel for Centronics devices, with an optional long lines interface for Versatec devices.

Devices attached to this interface are normally handled by the line printer spooling system and should not be accessed by the user directly.

Opening the device */dev/vp0* or */dev/lp0* may yield one of two errors: ENXIO indicates that the device is already in use; EIO indicates that the device is offline.

The Versatec printer/plotter operates in either print or plot mode. To set the printer into plot mode you should include <vcmd.h> and use the *ioctl*(2) call:

```
ioctl(f, VSETSTATE, plotmd);
```

where *plotmd* is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

When going back into print mode from plot mode you normally eject paper by sending it an EOT after putting into print mode:

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(vpc);
f = fileno(vpc);
ioctl(f, VSETSTATE, prtmd);
write(f, "\04", 1);
```

FILES

/dev/vp0
/dev/lp0

BUGS

If you use the standard I/O library on the Versatec, be sure to explicitly set a buffer using *setbuf*, since the library will not use buffered output by default, and will run very slowly.

NAME

win – Sun window system

SYNOPSIS

pseudo-device win*number*
pseudo-device dtop*number*

DESCRIPTION

The *win* pseudo-device accesses the system drivers supporting the Sun window system. *number*, in the device description line above, indicates the maximum number of windows supported by the system. *number* is set to 128 in the *GENERIC* system configuration file used to generate the kernel used in Sun systems as they are shipped. The *dtop* pseudo-device line indicates the number of separate "desktops" (frame buffers) that can be actively running the Sun window system at once. In the *GENERIC* file, this number is set to 4.

Each window in the system is represented by a */dev/win** device. The windows are organized as a tree with windows being subwindows of their parents, and covering/covered by their siblings. Each window has a position in the tree, a position on a display screen, an input queue, and information telling what parts of it are exposed.

The window driver multiplexes keyboard and mouse input among the several windows, tracks the mouse with a cursor on the screen, provides each window access to information about what parts of it are exposed, and notifies the manager process for a window when the exposed area of the window changes so that the window may repair its display.

Full information on the window system functions is given in the *Programmer's Reference Manual for SunWindows.*

FILES

/dev/win[0-9]
/dev/win[0-9][0-9]

SEE ALSO

Programmer's Reference Manual for SunWindows

NAME

xt – Xylogics 472 1/2 inch tape controller

SYNOPSIS — SUN-3

controller xtc0 at vme16d16 ? csr 0xee60 priority 3 vector xtintr 0x64
controller xtc1 at vme16d16 ? csr 0xee68 priority 3 vector xtintr 0x65
tape xt0 at xtc0 drive 0 flags 1
tape xt1 at xtc1 drive 0 flags 1

SYNOPSIS — SUN-2

controller xtc0 at mbio ? csr 0xee60 priority 3
controller xtc0 at vme16 ? csr 0xee60 priority 3 vector xtintr 0x64
controller xtc1 at mbio ? csr 0xee68 priority 3
controller xtc1 at vme16 ? csr 0xee68 priority 3 vector xtintr 0x65
tape xt0 at xtc0 drive 0 flags 1
tape xt1 at xtc1 drive 0 flags 1

DESCRIPTION

The Xylogics 472 tape controller controls Pertec-interface 1/2" tape drives such as the CDC Keystone III, providing a standard tape interface to the device, see mtio(4). This controller is used to support high speed or high density drives, which are not supported effectively by the older TapeMaster controller (tm(4)).

The flags field is used to control remote density select operation: a 0 specifies no remote density selection is to be attempted, a 1 specifies that the Pertec density-select line is used to toggle between high and low density; a 2 specifies that the Pertc speed-select line is used to toggle between high and low density. The default is 1, which is appropriate for the CDC Keystone III (92185) and the Telex 9250. In no case will the controller select among more than 2 densities.

SEE ALSO

mt(1), tar(1), tm(4), mtio(4)

NAME

xy – Disk driver for Xylogics SMD Disk Controllers

SYNOPSIS — SUN-3

controller xyc0 at vme16d16 ? csr 0xee40 priority 2 vector xyintr 0x48
controller xyc1 at vme16d16 ? csr 0xee48 priority 2 vector xyintr 0x49
disk xy0 at xyc0 drive 0
disk xy1 at xyc0 drive 1
disk xy2 at xyc1 drive 0
disk xy3 at xyc1 drive 1

The two **controller** lines given in the synopsis sections above specify the first and second Xylogics 450 SMD disk controller in a Sun system.

SYNOPSIS — SUN-2

controller xyc0 at vme16 ? csr 0xee40 priority 2 vector xyintr 0x48
controller xyc1 at vme16 ? csr 0xee48 priority 2 vector xyintr 0x49
controller xyc0 at mbio ? csr 0xee40 priority 2
controller xyc1 at mbio ? csr 0xee48 priority 2
disk xy0 at xyc0 drive 0
disk xy1 at xyc0 drive 1
disk xy2 at xyc1 drive 0
disk xy3 at xyc1 drive 1

The first two **controller** lines specify the first and second Xylogics 450 SMD disk controllers in a Sun-2/160 VMEbus based system. The third and fourth **controller** lines specify the first and second Xylogics 450 SMD disk controllers in a Sun-2/120 or a Sun-2/170 Multibus based system.

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, and so on. The standard device names begin with 'xy' followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call usually results in only one I/O operation; therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r'.

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek*(2) calls should specify a multiple of 512 bytes.

If **flags 0x1** is specified, the overlapped seeks feature for that drive is turned off. Note that to be effective, the flag must be set on all drives for a specific controller. This action is necessary for controllers with older firmware, which have bugs preventing overlapped seeks from working properly.

DISK SUPPORT

This driver handles all SMD drives by reading a label from sector 0 of the drive which describes the disk geometry and partitioning.

The **xy?a** partition is normally used for the root file system on a disk, the xy?b partition as a paging area, and the **xy?c** partition for pack-pack copying (it normally maps the entire disk). The rest of the disk is normally the **xy?g** partition.

FILES

/dev/xy[0-7][a-h]	block files
/dev/rxy[0-7][a-h]	raw files

SEE ALSO

dkio(4S)

Xylogics Model 450 Peripheral Processor SMD Disk Subsystem Maintenance and Reference Manual (Sun 800-1025-01)

DIAGNOSTICS

xyc*n*: self test error

Self test error in controller, see the Maintenance and Reference Manual.

xyc*n*: WARNING: *n* bit addresses

The controller is strapped incorrectly. Sun systems use 20-bit addresses for Multibus based systems and 24-bit addresses for VMEbus based systems. See the subsection on the Xylogics controller in the appropriate Sun *Hardware Installation Manual* for your machine(s) for instructions on how to set the jumpers on the 450.

xy*n*: unable to read bad sector info

The bad sector forwarding information for the disk could not be read.

xy*n* and xy*n* are of same type (*n*) with different geometries.

The 450 does not support mixing the drive types found on these units on a single controller.

xy*n*: initialization failed

The drive could not be successfully initialized.

xy*n*: unable to read label

The drive geometry/partition table information could not be read.

xy*n*: Corrupt label

The geometry/partition label checksum was incorrect.

xy*n*: offline

A drive ready status is no longer detected, so the unit has been logically removed from the system. If the drive ready status is restored, the unit will automatically come back online the next time it is accessed.

xyn*c*: *cmd how* (*msg*) blk #*n* abs blk #*n*

A command such as read or write encountered an error condition (how): either it *failed*, the controller was *reset*, the unit was *restored*, or an operation was *retry*'ed. The *msg* is derived from the error number given by the controller, indicating a condition such as "drive not ready", "sector not found" or "disk write protected". The *blk #* is the sector in error relative to the beginning of the partition involved. The *abs blk #* is the absolute block number of the sector in error. Some fields of the error message may be missing since the information is not always available.

BUGS

In raw I/O *read*(2) and *write*(2) truncate file offsets to 512-byte block boundaries, and *write*(2) scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read*(2), *write*(2) and *lseek*(2) should always deal in 512-byte multiples.

Older revisions of the firmware do not properly support overlapped seeks. This will only affect systems with multiple disks on a single controller. If a large number of "zero sector count" errors appear, you should use the **flags** field to disable overlapped seeks.

NAME

zs – zilog 8530 SCC serial comunications driver

SYNOPSIS — SUN-3

device zs0 at obio ? csr 0x20000 flags 3 priority 3
device zs1 at obio ? csr 0x00000 flags 0x103 priority 3

SYNOPSIS — SUN-2

device zs0 at virtual ? csr 0xeec800 flags 3 priority 3
device zs1 at virtual ? csr 0xeec000 flags 0x103 priority 3
device zs2 at mbmem ? csr 0x80800 flags 3 priority 3
device zs3 at mbmem ? csr 0x81000 flags 3 priority 3
device zs4 at mbmem ? csr 0x84800 flags 3 priority 3
device zs5 at mbmem ? csr 0x85000 flags 3 priority 3

DESCRIPTION

The Zilog 8530 provides 2 serial communication ports with full modem control in asynchronous mode. Each port behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Of the synopsis lines above, the line for zs0 specifies the serial I/O ports provided by the CPU board, the line for zs1 specifies the Video Board ports (which are used for keyboard and mouse), the lines for zs2 and zs3 specify the first and second ports on the first SCSI board in a system, and those for zs4 and zs5 specify the first and second ports provided by the second SCSI board in a system, respectively.

Bit *i* of flags may be specified to say that a line is not properly connected, and that the line *i* should be treated as hard-wired with carrier always present. Thus specifying "flags 0x2" in the specification of zs0 would cause line ttyb to be treated in this way.

To allow a single tty line to be connected to a modem and used for both incoming and outgoing calls, a special feature, controlled by the minor device number, has been added. Minor device numbers in the range 0 – 127 correspond directly to the normal tty lines and are named *tty**. Minor device numbers in the range 128 – 256 correspond to the same physical lines as those above (i.e. the same line as the minor device number minus 128) and are (conventionally) named *cua**. The *cua* lines are special in that they can be opened even when there is no carrier on the line. Once a *cua* line is opened, the corresponding tty line can not be opened until the *cua* line is closed. Also, if the *tty* line has been opened successfully (usually only when carrier is recognized on the modem) the corresponding *cua* line can not be opened. This allows a modem to be attached to */dev/ttya* (usually renamed to */dev/ttyd0*) and used for dialin (by enabling the line for login in */etc/ttys*) and also used for dialout (by *tip*(1C) or *uucp*(1C)) as */dev/cua0* when no one is logged in on the line. Note that the bit in the flags word in the config file (see above) must be zero for this line.

FILES

/dev/tty[a, b, s0-s3]
/dev/ttyd[0-9, a-f]
/dev/cua[0-9, a-f]

SEE ALSO

tty(4)

DIAGNOSTICS

zs*n c*: silo overflow.

The character input silo overflowed before it could be serviced.

NAME

file formats formats of files used by various programs

DESCRIPTION

This section describes formats of files used by various programs.

a.out	a.out(5)	assembler and link editor output format
acct	acct(5)	execution accounting file
addresses	aliases(5)	addresses and mailing lists for sendmail(8)
aliases	aliases(5)	addresses and mailing lists for sendmail(8)
ar	ar(5)	archive (library) file format
core	core(5)	format of memory image file
cpio	cpio(5)	format of cpio archive
crontab	crontab(5)	table of times to run periodic jobs
directory	dir(5)	format of directories
dump	dump(5)	incremental dump format
dumpdates	dump(5)	incremental dump format
environment	environ(5V)	user (process) environment
ethers	ethers(5)	Ethernet address to hostname database or YP domain
exports	exports(5)	NFS file systems being exported
fcntl	fcntl(5)	file control options
fs	fs(5)	format of file system volume
fspec	fspec(5)	format specification in text files
fstab	mntent(5)	static information about filesystems
ftpusers	ftpusers(5)	list of users prohibited by ftp
gettytab	gettytab(5)	simplified terminal configuration data base
group	group(5)	local host's group file
hosts	hosts(5)	host name data base or YP domain
hosts.equiv	hosts.equiv(5)	list of trusted hosts
inode	fs(5)	format of file system index node
lastlog	usracct(5)	login records
magic	magic(5)	file command's magic number file
mntent	mntent(5)	static information about filesystems
mtab	mtab(5)	currently mounted file system table
netgroup	netgroup(5)	list of network groups
networks	networks(5)	network name data base or YP domain
passwd	passwd(5)	password file or YP domain
phones	phones(5)	remote host phone number data base
plot	plot(5)	graphics interface
printcap	printcap(5)	printer capability data base
protocols	protocols(5)	protocol name data base or YP domain
rasterfile	rasterfile(5)	Sun's file format for raster images
remote	remote(5)	remote host description file
resolver	resolver(5)	configuration file for name server routines
rmtab	rmtab(5)	local file system mount statistics
rpc	rpc(5)	rpc program number data base
sccsfile	sccsfile(5)	format of sccs(1) history file
servers	servers(5)	Internet server data base
services	services(5)	service name data base or YP domain
statmon	statmon(5)	statd directory and file structures
tar	tar(5)	tape archive file format
term	term(5)	terminal driving tables for nroff

termcap	termcap(5)	terminal capability data base
tp	tp(5)	DEC/mag tape formats
ttys	ttys(5)	terminal initialization data
ttytype	ttytype(5)	data base of terminal types by port
types	types(5)	primitive system data types
utmp	usracct(5)	login records
uuencode	uuencode(5)	format of an encoded uuencode file
vfont	vfont(5)	font formats
vgrindefs	vgrindefs(5)	vgrind's language definition data base
wtmp	usracct(5)	login records
ypfiles	ypfiles(5)	the yellowpages database and directory structure

NAME

a.out – assembler and link editor output format

SYNOPSIS

#include <a.out.h> #include <stab.h> #include <nlist.h>

DESCRIPTION

A.out is the output format of the assembler *as*(1) and the link editor *ld*(1). The link editor makes *a.out* executable files if there were no errors and no unresolved external references. Layout information as given in the include file for the Sun system is:

```
/*
* Header prepended to each a.out file.
*/
struct exec {
        unsigned        short       a_machtype;     /* machine type */
        unsigned        short       a_magic;        /* magic number */
        unsigned                    a_text;         /* size of text segment */
        unsigned                    a_data;         /* size of initialized data */
        unsigned                    a_bss;          /* size of uninitialized data */
        unsigned                    a_syms;         /* size of symbol table */
        unsigned                    a_entry;        /* entry point */
        unsigned                    a_trsize;       /* size of text relocation */
        unsigned                    a_drsize;       /* size of data relocation */
};

#define     M_68010     1           /* runs on either MC68010 or MC68020 */
#define     M_68020     2           /* runs only on MC68020 */

#define     OMAGIC      0407        /* magic number for old impure format */
#define     NMAGIC      0410        /* magic number for read-only text */
#define     ZMAGIC      0413        /* magic number for demand load format */

#define     PAGSIZ      0x2000      /* page size - same for Sun-2 and Sun-3 */
#define     SEGSIZ      0x20000     /* segment size - same for Sun-2 and Sun-3 */

/*
* The following macros take exec structures as arguments. N_BADMAG(x) returns
* 0 if the file has a reasonable magic number.
*/
#define     N_BADMAG(x) \
            (((x).a_magic)!=OMAGIC && ((x).a_magic)!=NMAGIC && ((x).a_magic)!=ZMAGIC)

/*
* Offsets to text|symbols|strings.
*/
#define     N_TXTOFF(x) \
            ((x).a_magic==ZMAGIC ? 0 : sizeof (struct exec))
#define N_SYMOFF(x) \
            (N_TXTOFF(x) + (x).a_text+(x).a_data + (x).a_trsize+(x).a_drsize)
#define     N_STROFF(x) \
            (N_SYMOFF(x) + (x).a_syms)
/*
* Macros which take exec structures as arguments and tell where the
* various pieces will be loaded.
```

```
*/
#define         N_TXTADDR(x) PAGSIZ
#define         N_DATADDR(x) \
                (((x).a_magic==OMAGIC)? (N_TXTADDR(x)+(x).a_text) \
                : (SEGSIZ+((N_TXTADDR(x)+(x).a_text-1) & ~(SEGSIZ-1))))
#define N_BSSADDR(x)  (N_DATADDR(x)+(x).a_data)
```

The *a.out* file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). In the header the sizes of each section are given in bytes. The last three sections may be absent if the program was loaded with the '–s' option of *ld* or if the symbols and relocation have been removed by *strip*(1).

The machine type in the header indicates the type of hardware on which the object code may be executed. Sun-2 code may be executed on Sun-3 systems, but not vice versa. Program files predating release 3.0 are recognized by a machine type of 0.

If the magic number in the header is OMAGIC (0407), it means that this is a non-sharable text which is not to be write-protected, so the data segment is immediately contiguous with the text segment. This is rarely used. If the magic number is NMAGIC (0410) or ZMAGIC (0413), the data segment begins at the first segment boundary following the text segment, and the text segment is not writable by the program; other processes executing the same file will share the text segment. For ZMAGIC format, the text and data sizes must both be multiples of the page size, and the pages of the file will be brought into the running image as needed, and not pre-loaded as with the other formats. This is suitable for large programs and is the default format produced by *ld*(1). The macros N_TXTADDR, N_DATADDR, and N_BSSADDR give the memory addresses at which the text, data, and bss segments, respectively, will be loaded.

In the ZMAGIC format, the size of the header is included in the size of the text section; in other formats, it is not.

When an *a.out* file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized data), and a stack. For the ZMAGIC format, the header is loaded with the text segment; for other formats it is not.

Program execution begins at the address given by the value of the **a-entry** field. In all file types other than XMAGIC, the is the same as N_TXTADDR(x). In ZMAGIC files it is N_TXTADDR + sizeof(struct exec).

The stack starts at the highest possible location in the memory image, and grows downwards. The stack is automatically extended as required. The data segment is extended as requested by *brk*(2) or *sbrk*(2).

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins at the beginning of the file for ZMAGIC format or just after the header for the other formats. The N_TXTOFF macro returns this absolute file position when given the name of an exec structure as argument. The data segment is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first 4 bytes of the string table are not used for string storage, but rather contain the size of the string table; this size *includes* the 4 bytes; thus, the minimum string table size is 4.

RELOCATION

The value of a byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol is added to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
* Format of a relocation datum.
*/
struct relocation_info {
        int             r_address;      /* address which is relocated */
        unsigned        r_symbolnum:24,/* local symbol ordinal */
                        r_pcrel:1,      /* was relocated pc relative already */
                        r_length:2,     /* 0=byte, 1=word, 2=long */
                        r_extern:1,     /* does not include value of sym referenced */
                        :4;             /* nothing, yet */
};
```

There is no relocation information if a_trsize+a_drsize==0. If r_extern is 0, then r_symbolnum is actually a n_type for the relocation (that is, N_TEXT meaning relative to segment text origin.)

SYMBOL TABLE

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file as follows:

```
/*
* Format of a symbol table entry.
*/
struct nlist {
        union {
                        char            *n_name;        /* for use when in-memory */
                        long            n_strx;         /* index into file string table */
        } n_un;
        unsigned char   n_type;                         /* type flag, that is, N_TEXT etc; see below */
        char            n_other;
        short           n_desc;                         /* see <stab.h> */
        unsigned        n_value;                        /* value of this symbol (or adb offset) */
};
#define         n_hash          n_desc                  /* used internally by ld */

/*
* Simple values for n_type.
*/
#define         N_UNDF          0x0                     /* undefined */
#define         N_ABS           0x2                     /* absolute */
#define         N_TEXT          0x4                     /* text */
#define         N_DATA          0x6                     /* data */
#define         N_BSS           0x8                     /* bss */
#define         N_COMM          0x12                    /* common (internal to ld) */
#define         N_FN            0x1f                    /* file name symbol */

#define         N_EXT           01                      /* external bit, or'ed in */
#define         N_TYPE          0x1e                    /* mask for all the type bits */

/*
* Other permanent symbol table entries have some of the N_STAB bits set.
* These are given in <stab.h>
*/
```

```
#define     N_STAB      0xe0        /* if any of these bits set, don't discard */
```

In the *a.out* file a symbol's n_un.n_strx field gives an index into the string table. A n_strx value of 0 indicates that no name is associated with a particular symbol table entry. The field n_un.n_name can be used to refer to the symbol name only if the program sets this up using n_strx and appropriate data from the string table. Because of the union in the nlist declaration, it is impossible in C to statically initialize such a structure. If this must be done (as when using *nlist*(3)) the file **<nlist.h>** should be included, rather that **<a.out.h>**; this contains the declaration without the union.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

SEE ALSO

adb(1), as(1), cc(1V), dbx(1), ld(1), nm(1), pc(1), strip(1)

NAME

acct – execution accounting file

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

The *acct*(2) system call makes entries in an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
/*      @(#)acct.h 1.1 86/07/07 SMI; from UCB 6.1 83/07/29*/

/*
 * Accounting structures;
 * these use a comp_t type which is a 3 bits base 8
 * exponent, 13 bit fraction ''floating point'' number.
 */
typedef u_short comp_t;

struct  acct
{
        char        ac_comm[10];  /* Accounting command name */
        comp_t      ac_utime;     /* Accounting user time */
        comp_t      ac_stime;     /* Accounting system time */
        comp_t      ac_etime;     /* Accounting elapsed time */
        time_t      ac_btime;     /* Beginning time */
        short       ac_uid;       /* Accounting user ID */
        short       ac_gid;       /* Accounting group ID */
        short       ac_mem;       /* average memory usage */
        comp_t      ac_io;        /* number of disk IO blocks */
        dev_t       ac_tty;       /* control typewriter */
        char        ac_flag;      /* Accounting flag */
};

#define AFORK    0001           /* has executed fork, but no exec */
#define ASU      0002           /* used super-user privileges */
#define ACOMPAT  0004           /* used compatibility mode */
#define ACORE    0010           /* dumped core */
#define AXSIG    0020           /* killed by a signal */

#ifdef KERNEL
#ifdef SYSACCT
struct  acct        acctbuf;
struct  vnode       *acctp;
#else
#define acct()
#endif
#endif
```

If the process does an *execve*(2), the first 10 characters of the filename appear in *ac_comm*. The accounting flag contains bits indicating whether *execve*(2) was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), execve(2), sa(8)

NAME

aliases, addresses, forward – addresses and aliases for sendmail(8)

SYNOPSIS

/etc/passwd
/usr/lib/aliases
/usr/lib/aliases.dir
/usr/lib/aliases.pag
~/.forward

DESCRIPTION

These files contain mail addresses or aliases recognized by *sendmail*(8):

/etc/passwd Mail addresses (usernames) of local users.

/usr/lib/aliases Local aliases in ASCII format. This file can be edited to add, update, or delete mail aliases for the local host.

/usr/lib/aliases.{dir,pag}
The aliasing information from */usr/lib/aliases*, in binary, *dbm*(3X) format for use by *sendmail*(8). The program *newaliases*(8), which is invoked automatically by *sendmail*(8), maintains these files.

~/.forward Addresses to which a user's mail is forwarded (see *Automatic Forwarding*, below).

In addition, the Yellow Pages aliases map *mail.aliases* contains addresses and aliases available for use across the network.

ADDRESSES

As distributed, *sendmail*(8) supports the following types of addresses:

- Local usernames. These are listed in the local host's */etc/passwd* file.
- Local filenames. When mailed to an absolute pathname, a message can be appeneded to a file.
- Commands. If the first character of the address is a vertical bar, (|), *sendmail (8)* pipes the message to the standard input of the command the bar precedes.
- DARPA-standard mail addresses of the form:

  name@domain

  If *domain* does not contain any dots (.), then it is interpreted as the name of a host in the current domain. Otherwise, the message is passed to a *mailhost* that determines how to get to the specified domain. Domains are divided into subdomains separated by dots, with the top-level domain on the right. Top-level domains include:

  .COM Commerical organizations.
  .EDU Educational organizations.
  .GOV Government organizations.
  .MIL Military organizations.

  For example, the full address of John Smith could be:

  js@jsmachine.Podunk-U.EDU

  if he uses the machine named "jsmachine" at Podunk University.

- *uucp*(1C) addresses of the form:

  ... [*host*!]*host*!*username*

  These are sometimes mistakenly referred to as "Usenet" addresses. *uucp*(1C) provides links to numerous sites throughout the world for the remote copying of files.

Other site-specific forms of addressing can be added by customizing the *sendmail* configuration file. See the *sendmail*(8), and *Sendmail Installation and Operation* in *System Administration for the Sun Workstation* for details. Standard addresses are recommended.

ALIASES

Local Aliases

/usr/lib/aliases is formatted as a series of lines of the form

name: *address*[, *address*]

name is the name of the alias or alias group, and *address* is the address of a recipient in the group. Aliases can be nested. That is, an *address* can be the name of another alias group. Lines beginning with white space are treated as continuation lines for the preceding alias. Lines beginning with # are comments.

Special Aliases

An alias of the form:

owner–*aliasname*: *address*

directs error-messages resulting from mail to *alias-name* to *address*, instead of back to the person who sent the message.

An alias of the form:

aliasname: **:include:***pathname*

with colons as shown, adds the recipients listed in the file *pathname* to the *aliasname* alias. This allows a private list to be maintained separately from the aliases file.

YP Domain Aliases

Normally, the aliases file on the master YP server is used for the *mail.aliases* YP map, which can be made available to every YP client. Thus, the */usr/lib/aliases** files on the various hosts in a network will one day be obsolete. Domain-wide aliases should ultimately into usernames on specific hosts. For example, if the following were in the domain-wide alias file:

jsmith:js@jsmachine

then any YP client could just mail to "jsmith" and not have to remember the machine and user name for John Smith. If a YP alias does not resolve to an address with a specific host, then the name of the YP domain is used. There should be an alias of the domain name for a host in this case. For example, the alias:

jsmith:root

sends mail on a YP client to "root@podunk-u" if the name of the YP domain is "podunk-u".

Automatic Forwarding

When an alias (or address) is resolved to a the name of a user on the local host, *sendmail* checks for a *.forward* file in that user's home directory. This file can contain one or more addresses or aliases as described above; each recipient is sent a copy of the mail destined for the original user.

Care must be taken to avoid creating addressing loops in the *.forward* file. When forwarding mail between machines, be sure that the destination machine does not return the mail to the sender through the operation of any YP aliases. Otherwise, copies of the message may "bounce." Usually, the solution is to change the YP alias to direct mail to the proper destination.

A backslash before a username inhibits further aliasing. For instance, to invoke the *vacation*(1) program, user **js** creates a *.forward* file that contains the line:

```
\js, "|/usr/ucb/vacation js"
```

so that one copy of the message is sent to the user, and another is piped into the *vacation*(1) program.

SEE ALSO

newaliases(8), dbm(3X), sendmail(8), uucp(1C), vacation(1)

System Administration for the Sun Workstation

BUGS

Because of restrictions in *dbm*(3X) a single alias cannot contain more than about 1000 characters. Nested aliases can be used to circumvent this limit.

NAME

ar – archive (library) file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar* combines several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
/*      @(#)ar.h 1.1 86/07/07 SMI; from UCB 4.1 83/05/03*/

#define ARMAG   "!<arch>\n"
#define SARMAG 8

#define ARFMAG "`\n"

struct ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_fmag[2];
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode*, which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on a even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

SEE ALSO

ar(1), ld(1), nm(1)

BUGS

File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

NAME

core – format of memory image file

SYNOPSIS

#include <sys/core.h>

DESCRIPTION

The UNIX System writes out a memory image of a terminated process when any of various errors occur. See *sigvec*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The maximum size of a *core* file is limited by *setrlimit*(2). Files which would be larger than the limit are not created.

Set-user-id programs do not produce core files when they terminate as this would be a security loophole.

The core file consists of a **core** structure defined in the *<sys/core.h>* file. The **core** structure includes the registers, the floating point status, the program's header, the size of the text, data, and stack segments, the name of the program and the number of the signal that terminated the process. The program's header is described by the **exec** structure defined in the *<sys/exec.h>* file.

The remainder of the core file consists first of the data pages and then the stack pages of the process image. The amount of data space image in the core file is given (in bytes) by the *c_dsize* member of the **core** structure. The amount of stack image in the core file is given (in bytes) by the *c_ssize* member of the **core** structure.

SEE ALSO

adb(1), dbx(1), sigvec(2), setrlimit(2)

NAME

cpio – format of cpio archive

DESCRIPTION

The old format *header* structure, when the –c option of *cpio* is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to a word];
} Hdr;
```

The byte order here is that of the machine on which the tape was written. If the tape is being read on a machine with a different byte order, you have to use *swab*(3) after reading the header. You can determine what byte order the tape was written with by examining the *h_magic* field; if it is equal to 0143561 (octal), which is the standard magic number 070707 (octal) with the bytes swapped, the tape was written in a byte order opposite to that of the machine on which it is being read. If you are producing a tape to be read on a machine with the opposite byte order to that of the machine on which it is being produced, you can use *swap* before writing the header.

When the –c option is used, the *header* information is described by the statement below:

```
sscanf(Chdr, "%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Hdr.h_mtime, &Hdr.h_namesize, &Hdr.h_filesize, &Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer, are recorded with *h_filesize* equal to zero. Symbolic links are recorded similarly to regular files, with the "contents" of the file being the name of the file the symbolic link points to.

SEE ALSO

cpio(1), find(1), stat(2)

NAME

crontab – table of times to run periodic jobs

SYNOPSIS

/usr/lib/crontab

DESCRIPTION

The */etc/cron* utility is a permanent process, started by */etc/rc.local*, that wakes up once every minute. */etc/cron* consults the file */usr/lib/crontab* to find out what tasks are to be done, and at what time.

Each line in */usr/lib/crontab* consists of six fields, separated by spaces or tabs, as follows:

1. minutes field, which can have values in the range 0 through 59.
2. hours field, which can have values in the range 0 through 23.
3. day of the month, in the range 1 through 31.
4. month of the year, in the range 1 through 12.
5. day of the week, in the range 1 through 7. Monday is day 1 in this scheme of things.
6. (the remainder of the line) is the command to be run. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Any of fields 1 through 5 can be a list of values separated by commas. A field can be a pair of numbers separated by a hyphen, indicating that the job is to be done for all the times in the specified range. If a field is an asterisk character (*) it means that the job is done for all possible values of the field.

FILES

/usr/lib/crontab

SEE ALSO

cron(8), rc(8)

EXAMPLE

```
0 0 * * * calendar -
15 0 * * * /etc/sa -s >/dev/null
15 4 * * * find /usr/preserve -mtime +7 -a -exec rm -f {} ;
40 4 * * * find / -name '#*' -atime +3 -exec rm -f {} ;
0,15,30,45 * * * * /etc/atrun
0,10,20,30,40,50 * * * * /etc/dmesg - >>/usr/adm/messages
5 4 * * * sh /etc/newsyslog
```

The *calendar* command run at minute 0 of hour 0 (midnight) of every day. The */etc/sa* command runs at 15 minutes after midnight every day. The two *find* commands run at 15 minutes past four and at 40 minutes past four, respectively, every day of the year. The *atrun* command (which processes shell scripts users have set up with *at*) runs every 15 minutes. The */etc/dmesg* command appends kernel messages to the */usr/adm/messages* file every ten minutes, and finally, the */usr/adm/syslog* script runs at five minutes after four every day.

NAME

dir – format of directories

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>
```

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory and directories must be read using the *getdirentries*(2) system call or the *directory*(3) library routines. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry; see *fs*(5).

A directory consists of some number of blocks of DIRBLKSIZ bytes, where DIRBLKSIZ is chosen such that it can be transferred to disk in a single atomic operation (512 bytes on most machines):

```
#ifdef KERNEL
#define DIRBLKSIZ DEV_BSIZE
#else
#define DIRBLKSIZ 512
#endif

#define MAXNAMLEN 255
```

Each DIRBLKSIZ byte block contains some number of directory entry structures, which are of variable length. Each directory entry has a **struct direct** at the front of it, containing its inode number, the length of the entry, and the length of the name contained in the entry. These are followed by the name padded to a 4-byte boundary with null bytes. All names are guaranteed null terminated. The maximum length of a name in a directory is MAXNAMLEN.

The macro **DIRSIZ(dp)** gives the amount of space required to represent a directory entry. Free space in a directory is represented by entries that have:

```
dp->d_reclen > DIRSIZ(dp)
```

All DIRBLKSIZ bytes in a directory block are claimed by the directory entries. This usually results in the last entry in a directory having a large **dp->d_reclen**. When entries are deleted from a directory, the space is returned to the previous entry in the same directory block by increasing its **dp->d_reclen**. If the first entry of a directory block is free, then its **dp->d_ino** is set to 0. Entries other than the first in a directory do not normally have **dp->d_ino** set to 0.

The DIRSIZ macro gives the minimum record length which will hold the directory entry. This requires the amount of space in struct direct without the **d_name** field, plus enough space for the name with a terminating null byte **(dp->d_namlen+1)**, rounded up to a 4-byte boundary.

```
#undef DIRSIZ
#define DIRSIZ(dp)    ((sizeof (struct direct) - (MAXNAMLEN+1)) + (((dp)->d_namlen+1 + 3) &~ 3))
struct    direct {
        u_long  d_ino;
        short   d_reclen;
        short   d_namlen;
        char    d_name[MAXNAMLEN + 1];
        /* typically shorter */
};

struct _dirdesc {
        int     dd_fd;
        long    dd_loc;
        long    dd_size;
        char    dd_buf[DIRBLKSIZ];
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system (''/''), for which '..' has the same meaning as '.'.

SEE ALSO

fs(5), readdir(3)

NAME

dump, dumpdates – incremental dump format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/inode.h>
#include <dumprestor.h>
```

DESCRIPTION

Tapes used by *dump* and *restore*(8) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file *<dumprestor.h>* is:

```
#define NTREC           10
#define MLEN    16
#define MSIZ    4096

#define TS_TAPE         1
#define TS_INODE        2
#define TS_BITS         3
#define TS_ADDR         4
#define TS_END          5
#define TS_CLRI         6
#define MAGIC           (int) 60011
#define CHECKSUM        (int) 84446

struct  spcl {
        int             c_type;
        time_t          c_date;
        time_t          c_ddate;
        int             c_volume;
        daddr_t         c_tapea;
        ino_t           c_inumber;
        int             c_magic;
        int             c_checksum;
        struct          dinode          c_dinode;
        int             c_count;
        char            c_addr[BSIZE];
} spcl;

struct  idates {
        char            id_name[16];
        char            id_incno;
        time_t          id_ddate;
};

#define  DUMPOUTFMT "%-16s %c %s"           /* for printf */
                                            /* name, incno, ctime(date) */
#define  DUMPINFMT   "%16s %c %[^\n]\n"     /* inverse for scanf */
```

NTREC is the default number of 1024 byte records in a physical tape block, changeable by the **b** option to *dump*. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See *c_addr* below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in *c_magic*.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (1024-byte) record.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system; see *fs*(5).
c_count	The count of characters in *c_addr*.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END record and then the tapemark.

The structure *idates* describes an entry in the file */etc/dumpdates* where dump history is kept. The fields of the structure are:

id_name	The dumped filesystem is '/dev/*id_nam*'.
id_incno	The level number of the dump tape; see *dump*(8).
id_ddate	The date of the incremental dump in system format see *types*(5).

FILES

/etc/dumpdates

SEE ALSO

dump(8), restore(8), fs(5), types(5)

BUGS

Should more explicitly describe format of dumpdates file.

NAME

environ – user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

An array of strings called the 'environment' is made available by *execve*(2) when a process begins. By convention these strings have the form '*name=value*'. The following names are used by various commands:

PATH — The sequence of directory prefixes that *sh*, *time*, *nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). The *login*(1) process sets **PATH=:/usr/ucb:/bin:/usr/bin.**

HOME — The name of the user's login directory, set by *login*(1) from the password file */etc/passwd* (see *passwd*(5)).

TERM — The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*(1G), which may exploit special terminal capabilities. See */etc/termcap* (*termcap*(5)) for a list of terminal types.

SHELL — The file name of the user's login shell.

TERMCAP — The string describing the terminal in TERM, or the name of the termcap file, see *termcap*(3),*termcap*(5),

EXINIT — A startup list of commands read by *ex*(1), *edit*(1), and *vi*(1).

USER
LOGNAME — The login name of the user.

Further names may be placed in the environment by the *export* command and 'name=value' arguments in *sh*(1), or by the *setenv* command if you use *csh*(1). Arguments may also be placed in the environment at the point of an *execve*(2). It is unwise to conflict with certain *sh*(1) variables that are frequently exported by **.profile** files: MAIL, PS1, PS2, IFS.

SYSTEM V DESCRIPTION

The description of the variable TERMCAP does not apply to the System V environment.

TZ — Time zone information. The format is *xxx n zzz* where *xxx* is standard local time zone abbreviation, *n* is the difference in hours from GMT, and *zzz* is the abbreviation for the daylight-saving local time zone, if any; for example, **EST 5 EDT**.

SEE ALSO

csh(1), ex(1), login(1), sh(1), getenv(3), execve(2), system(3), termcap(3X), termcap(5)

NAME

ethers – Ethernet address to hostname database or YP domain

DESCRIPTION

The *ethers* file contains information regarding the known (48 bit) Ethernet addresses of hosts on the Internet. For each host on an Ethernet, a single line should be present with the following information:

Ethernet address
official host name

Items are separated by any number of blanks and/or tabs. A '#' indicates the beginning of a comment extending to the end of line.

The standard form for Ethernet addresses is "x:x:x:x:x:x" where *x* is a hexadecimal number between 0 and ff, representing one byte. The address bytes are always in network order. Host names may contain any printable character other than a space, tab, newline, or comment character. It is intended that host names in the *ethers* file correspond to the host names in the *hosts*(5) file.

The *ether_line*() routine from the Ethernet address manipulation library, *ethers*(3N) may be used to scan lines of the *ethers* file.

FILES

/etc/ethers

SEE ALSO

ethers(3N), hosts(5)

NAME

exports – NFS file systems being exported

SYNOPSIS

/etc/exports

DESCRIPTION

The file */etc/exports* describes the file systems which are being exported to nfs clients. It is created by the system administrator using a text editor and processed by the *mount* request daemon *mountd*(8C) each time a mount request is received.

The file consists of a list of file systems and the *netgroup*(5) or machine names allowed to remote mount each file system. The file system names are left justified and followed by a list of names separated by white space. The names will be looked up in */etc/netgroup* and then in */etc/hosts*. A file system name with no name list following means export to everyone. A '#' anywhere in the file indicates a comment extending to the end of the line it appears on. Lines beginning with white space are continuation lines.

EXAMPLE

```
/usr        clients                 # export to my clients
/usr/local                          # export to the world
/usr2       phoenix sun sundae      # export to only these machines
```

FILES

/etc/exports

SEE ALSO

mountd(8C)

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2V) as shown below:

```
/*      @(#)fcntl.h 1.2 83/12/08 SMI; from UCB 4.2 83/09/25      */

/*
 * Flag values accessible to open(2V) and fcntl(2)
 * (The first three can only be set by open)
 */
#define O_RDONLY    0
#define O_WRONLY    1
#define O_RDWR              2
#define O_NDELAY    FNDELAY     /* Non-blocking I/O */
#define O_APPEND    FAPPEND     /* append (writes guaranteed at the end) */

#ifndef F_DUPFD
/* fcntl(2) requests */
#define F_DUPFD     0       /* Duplicate fildes */
#define F_GETFD     1       /* Get fildes flags */
#define F_SETFD     2       /* Set fildes flags */
#define F_GETFL     3       /* Get file flags */
#define F_SETFL     4       /* Set file flags */
#define F_GETOWN 5  /* Get owner */
#define F_SETOWN 6  /* Set owner */

/* flags for F_GETFL, F_SETFL-- copied from <sys/file.h> */
#define FNDELAY             00004       /* non-blocking reads */
#define FAPPEND             00010       /* append on each write */
#define FASYNC              00100       /* signal pgrp when data ready */
#endif
```

SEE ALSO

fcntl(2), open(2V)

NAME

fs, inode – format of file system volume

SYNOPSIS

#include <sys/types.h>
#include <sys/filsys.h>
#include <sys/inode.h>

DESCRIPTION

Every file system storage volume (disk, nine-track tape, for instance) has a common format for certain vital information. Every such volume is divided into a certain number of blocks. The block size is a parameter of the file system. Sectors 0 to 15 on a file system are used to contain primary and secondary bootstrapping programs.

The actual file system begins at sector 16 with the *super block*. The layout of the super block as defined by the include file *<sys/fs.h>* is:

```
#define FS_MAGIC      0x011954
struct   fs {
        struct   fs *fs_link;           /* linked list of file systems */
        struct   fs *fs_rlink;          /*    used for incore super blocks */
        daddr_t fs_sblkno;              /* addr of super-block in filesys */
        daddr_t fs_cblkno;              /* offset of cyl-block in filesys */
        daddr_t fs_iblkno;              /* offset of inode-blocks in filesys */
        daddr_t fs_dblkno;              /* offset of first data after cg */
        long    fs_cgoffset;            /* cylinder group offset in cylinder */
        long    fs_cgmask;              /* used to calc mod fs_ntrak */
        time_t  fs_time;                /* last time written */
        long    fs_size;                /* number of blocks in fs */
        long    fs_dsize;       /* number of data blocks in fs */
        long    fs_ncg;                 /* number of cylinder groups */
        long    fs_bsize;       /* size of basic blocks in fs */
        long    fs_fsize;               /* size of frag blocks in fs */
        long    fs_frag;                /* number of frags in a block in fs */
/* these are configuration parameters */
        long    fs_minfree;             /* minimum percentage of free blocks */
        long    fs_rotdelay;            /* num of ms for optimal next block */
        long    fs_rps;                 /* disk revolutions per second */
/* these fields can be computed from the others */
        long    fs_bmask;               /* ''blkoff'' calc of blk offsets */
        long    fs_fmask;               /* ''fragoff'' calc of frag offsets */
        long    fs_bshift;              /* ''lblkno'' calc of logical blkno */
        long    fs_fshift;              /* ''numfrags'' calc number of frags */
/* these are configuration parameters */
        long    fs_maxcontig;           /* max number of contiguous blks */
        long    fs_maxbpg;              /* max number of blks per cyl group */
/* these fields can be computed from the others */
        long    fs_fragshift;           /* block to frag shift */
        long    fs_fsbtodb;             /* fsbtodb and dbtofsb shift constant */
        long    fs_sbsize;              /* actual size of super block */
        long    fs_csmask;              /* csum block offset */
        long    fs_csshift;             /* csum block number */
        long    fs_nindir;              /* value of NINDIR */
        long    fs_inopb;               /* value of INOPB */
        long    fs_nspf;        /* value of NSPF */
        long    fs_optim;               /* optimization preference, see below */
```

```
        long    fs_sparecon[3];         /* reserved for future constants */
/* a unique id for this filesystem (currently unused and unmaintained) */
        long    fs_id[2];               /* file system id */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;              /* blk addr of cyl grp summary area */
        long    fs_cssize;              /* size of cyl grp summary area */
        long    fs_cgsize;              /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;               /* tracks per cylinder */
        long    fs_nsect;               /* sectors per track */
        long    fs_spc;                 /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;                /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;                 /* cylinders per group */
        long    fs_ipg;                 /* inodes per group */
        long    fs_fpg;                 /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct  csum fs_cstotal;  /* cylinder summary information */
/* these fields are cleared at mount time */
        char    fs_fmod;                /* super block modified flag */
        char    fs_clean;               /* file system is clean flag */
        char    fs_ronly;               /* mounted read-only flag */
        char    fs_flags;               /* currently unused flag */
        char    fs_fsmnt[MAXMNTLEN];        /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;             /* last cg searched */
        struct  csum *fs_csp[MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;                 /* cyl per cycle in postbl */
        short   fs_postbl[MAXCPG][NRPOS];/* head of blocks for each rotation */
        long    fs_magic;               /* magic number */
        u_char  fs_rotbl[1];            /* list of blocks for each rotation */
/* actually longer */
};
```

Each disk drive contains some number of file systems. A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of 'blocks'. File system blocks of at most size MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated as only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode is the root of the file system. Inode 0 can't be used for normal purposes and historically bad blocks were linked to inode 1, thus the root inode is 2 (inode 1 is no longer used for this purpose, however numerous dump tapes make this assumption, so we are stuck with it). The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

fs_minfree gives the minimum acceptable percentage of file system blocks which may be free. If the freelist drops below this level only the super-user may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 90% comes with a fragmentation of 4, thus the default fragment size is a fourth of the block size.

Cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. With NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms.

Each file system has a statically allocated number of inodes. An inode is allocated for each NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having the only a single variable size element (the free bit map).

N.B.: MAXIPG must be a multiple of INOPB(fs).

MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus changes to (struct cg) must keep its size within MINBSIZE. MAXCPG is limited only to dimension an array in (struct cg); it can be made larger as long as that structure's size remains within the bounds dictated by MINBSIZE. Note that super blocks are never more than size SBSIZE.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

Super block for a file system: MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size SBSIZE. The size of these tables is **inversely** proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

Inode: The inode is the focus of all file activity in the UNIX file system. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is 'named' by its device/i-number pair. For further information, see the include file <sys/inode.h>.

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets **<:** and **:>**. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are as follows:

a 1,10,16,36,72
Assembler, IBM S/370, first format

a2 1,10,16,40,72
Assembler, IBM S/370, second format

c 1,8,12,16,20,55
COBOL, normal format

c2 1,6,10,14,49
COBOL compact format (columns 1-6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows:

<:t–c2 m6 s66 d:>

c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than **c2**. This is the recommended format for COBOL. The appropriate format specification is:

<:t–c3 m6 s66 d:>

f 1,7,11,15,19,23
FORTRAN

p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

s 1,10,55
SNOBOL

u 1,12,20,44
UNIVAC 1100 Assembler

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin*

The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several UNIX system commands correctly interpret the format specification for a file. Among them is *newform*(1) which may be used to convert files to a standard format acceptable to other UNIX system commands.

SEE ALSO

newform(1)

NAME

ftpusers – list of users prohibited by ftp

SYNOPSIS

/usr/etc/ftpusers

DESCRIPTION

Ftpusers contains a list of users who cannot access this system using the *ftp*(1) program. *Ftpusers* contains one user name per line.

SEE ALSO

ftp(1C), ftpd(8C)

NAME

gettytab – terminal configuration data base

SYNOPSIS

/etc/gettytab

DESCRIPTION

Gettytab is a simplified version of the *termcap*(5) data base used to describe terminal lines. The initial terminal login process *getty*(8) accesses the *gettytab* file each time it starts, allowing simpler reconfiguration of terminal characteristics. Each entry in the data base is used to describe one class of terminals.

There is a default terminal class, *default*, that is used to set global defaults for all other classes. (That is, the *default* entry is read, then the entry for the class required is used to override particular settings.)

CAPABILITIES

Refer to *termcap*(5) for a description of the file layout. The *default* column below lists defaults obtained if there is no entry in the table obtained, nor one in the special *default* table.

Name	Type	Default	Description
ap	bool	false	terminal uses any parity
bd	num	0	backspace delay
bk	str	0377	alternate end of line character (input break)
cb	bool	false	use crt backspace mode
cd	num	0	carriage-return delay
ce	bool	false	use crt erase algorithm
ck	bool	false	use crt kill algorithm
cl	str	NULL	screen clear sequence
co	bool	false	console - add \n after login prompt
ds	str	^Y	delayed suspend character
ec	bool	false	leave echo OFF
ep	bool	false	terminal uses even parity
er	str	^?	erase character
et	str	^D	end of text (EOF) character
ev	str	NULL	initial enviroment
f0	num	unused	tty mode flags to write messages
f1	num	unused	tty mode flags to read login name
f2	num	unused	tty mode flags to leave terminal as
fd	num	0	form-feed (vertical motion) delay
fl	str	^O	output flush character
hc	bool	false	do NOT hangup line on last close
he	str	NULL	hostname editing string
hn	str	hostname	hostname
ht	bool	false	terminal has real tabs
ig	bool	false	ignore garbage characters in login name
im	str	NULL	initial (banner) message
in	str	^C	interrupt character
is	num	unused	input speed
kl	str	^U	kill character
lc	bool	false	terminal has lower case
lm	str	login:	login prompt
ln	str	^V	''literal next'' character
lo	str	/bin/login	program to exec when name obtained
nd	num	0	newline (line-feed) delay
nl	bool	false	terminal has (or might have) a newline character
nx	str	default	next table (for auto speed selection)

op	bool	false	terminal uses odd parity
os	num	unused	output speed
pc	str	\0	pad character
pe	bool	false	use printer (hard copy) erase algorithm
pf	num	0	delay between first prompt and following flush (seconds)
ps	bool	false	line connected to a MICOM port selector
qu	str	^\	quit character
rp	str	^R	line retype character
rw	bool	false	do NOT use raw for input, use cbreak
sp	num	0	line speed (input and output)
su	str	^Z	suspend character
tc	str	none	table continuation
td	num	0	tab delay
to	num	0	timeout (seconds)
tt	str	NULL	terminal type (for enviroment)
ub	bool	false	do unbuffered output (of prompts etc)
uc	bool	false	terminal is known upper case only
we	str	^W	word erase character
xc	bool	false	do NOT echo control chars as ^X
xf	str	^S	XOFF (stop output) character
xn	str	^Q	XON (start output) character

If no line speed is specified, speed will not be altered from that which prevails when *getty* is entered. Specifying an input or output speed overrides line speed for stated direction only.

Terminal modes to be used for the output of the message, for input of the login name, and to leave the terminal set as upon completion, are derived from the Boolean flags specified. If the derivation should prove inadequate, any (or all) of these three may be overriden with one of the **f0**, **f1**, or **f2** numeric specifications, which can be used to specify (usually in octal, with a leading '0') the exact values of the flags. Local (new tty) flags are set in the top 16 bits of this (32 bit) value.

Should *getty* receive a null character (presumed to indicate a line break) it will restart using the table indicated by the **nx** entry. If there is none, it will re-use its original table.

Delays are specified in milliseconds, the nearest possible delay available in the tty driver will be used. Should greater certainty be desired, delays with values 0, 1, 2, and 3 are interpreted as choosing that particular delay algorithm from the driver.

The **cl** screen clear string may be preceded by a (decimal) number of milliseconds of delay required (a la termcap). This delay is simulated by repeated use of the pad character **pc**.

The initial message, and login message, **im** and **lm** may include the character sequence **%h** to obtain the hostname. (**%%** obtains a single '%' character.) The hostname is normally obtained from the system, but may be set by the **hn** table entry. In either case it may be edited with **he**. The **he** string is a sequence of characters, each character that is neither '@' nor '#' is copied into the final hostname. A '@' in the **he** string, causes one character from the real hostname to be copied to the final hostname. A '#' in the **he** string, causes the next character of the real hostname to be skipped. Surplus '@' and '#' characters are ignored.

When *getty* execs the login process, given in the **lo** string (usually "/bin/login"), it will have set the enviroment to include the terminal type, as indicated by the **tt** string (if it exists). The **ev** string, can be used to enter additional data into the environment. It is a list of comma separated strings, each of which will presumably be of the form *name=value*.

If a non-zero timeout is specified, with **to**, then *getty* will exit within the indicated number of seconds, either having received a login name and passed control to *login*, or having received an alarm signal, and exited. This may be useful to hangup dial in lines.

Output from *getty* is even parity unless **op** is specified. **Op** may be specified with **ap** to allow any parity on input, but generate odd parity output. Note: this only applies while *getty* is being run, terminal driver limitations prevent a more complete implementation. *Getty* does not check parity of input characters in *RAW* mode.

FILES

/etc/gettytab

SEE ALSO

termcap(5), getty(8).

NAME

group – group file

SYNOPSIS

/etc/group

DESCRIPTION

Group contains for each group the following information:

- group name
- encrypted password
- numerical group ID
- a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a newline. If the password field is null, no password is demanded.

This file resides in the */etc* directory. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

A group file can have a line beginning with a plus (+), which means to incorporate entries from the yellow pages. There are two styles of + entries: All by itself, + means to insert the entire contents of the yellow pages group file at that point; *+name* means to insert the entry (if any) for *name* from the yellow pages at that point. If a + entry has a non-null password or group member field, the contents of that field will overide what is contained in the yellow pages. The numerical group ID field cannot be overridden.

EXAMPLE

```
+myproject:::bill, steve
+:
```

If these entries appear at the end of a group file, then the group *myproject* will have members *bill* and *steve*, and the password and group ID of the yellow pages entry for the group *myproject*. All the groups listed in the yellow pages will be pulled in and placed after the entry for *myproject*.

FILES

/etc/group /etc/yp/group

SEE ALSO

setgroups(2), initgroups(3), crypt(3), passwd(1), passwd(5)

BUGS

The *passwd*(1) command won't change group passwords.

NAME

hosts – host name data base

SYNOPSIS

/etc/hosts

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

Internet address
official host name
aliases

Items are separated by any number of blanks and/or tab characters. A '#' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional '.' notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

Here is a typical line from the */etc/hosts* file:

```
192.9.1.20      gaia            # Alison Shanks
```

FILES

/etc/hosts

SEE ALSO

gethostent(3N)

NAME

hosts.equiv – list of trusted hosts

DESCRIPTION

Hosts.equiv resides in directory */etc* and contains a list of trusted hosts. When an *rlogin*(1) or *rsh*(1) request from such a host is made, and the initiator of the request is in */etc/passwd*, then no further validity checking is done. That is, *rlogin* does not prompt for a password, and *rsh* completes successfully. So a remote user is "equivalenced" to a local user with the same user ID when the remote user is in *hosts.equiv*.

The format of *hosts.equiv* is a list of names, as in this example:

```
host1
host2
+@group1
-@group2
```

A line consisting of a simple host name means that anyone logging in from that host is trusted. A line consisting of *+@group* means that all hosts in that network group are trusted. A line consisting of *–@group* means that hosts in that group are not trusted. Programs scan *hosts.equiv* linearly, and stop at the first hit (either positive for hostname and +@ entries, or negative for –@ entries). A line consisting of a single + means that everyone is trusted.

The *.rhosts* file has the same format as *hosts.equiv*. When user *XXX* executes *rlogin* or *rsh*, the *.rhosts* file from *XXX*'s home directory is conceptually concatenated onto the end of *hosts.equiv* for permission checking. However, –@ entries are not sticky. If a user is excluded by a minus entry from *hosts.equiv* but included in *.rhosts*, then that user is considered trusted. In the special case when the user is root, then only the */.rhosts* file is checked.

It is also possible to have two entries (separated by a single space) on a line of these files. In this case, if the remote host is equivalenced by the first entry, then the user named by the second entry is allowed to log in as anyone, that is, specify any name to the **–l** flag (provided that name is in the */etc/passwd* file, of course). Thus

```
sundown john
```

allows *john* to log in from sundown as anyone. The usual usage would be to put this entry in the *.rhosts* file in the home directory for *bill* . Then *john* may log in as *bill* when coming from sundown. The second entry may be a netgroup, thus

```
+@group1 +@group2
```

allows any user in *group2* coming from a host in *group1* to log in as anyone.

FILES

/etc/hosts.equiv

SEE ALSO

rlogin(1), rsh(1), netgroup(5)

NAME

magic – file command's magic number file

DESCRIPTION

The *file*(1) command identifies the type of a file using, among other tests, a test for whether the file begins with a certain *magic number*. The file **/etc/magic** specifies what magic numbers are to be tested for, what message to print if a particular magic number is found, and additional information to extract from the file.

Each line of the file specifies a test to be performed. A test compares the data starting at a particular offset in the file with a 1-byte, 2-byte, or 4-byte numeric value or a string. If the test succeeds, a message is printed. The line consists of the following fields:

offset A number specifying the offset, in bytes, into the file of the data which is to be tested.

type The type of the data to be tested. The possible values are:

byte A one-byte value.

short A two-byte value.

long A four-byte value.

string A string of bytes.

The types **byte**, **short**, and **long** may optionally be followed by a mask specifier of the form *&number*. If a mask specifier is given, the value is AND'ed with the *number* before any comparisons are done. The *number* is specified in C form; e.g. **13** is decimal, **013** is octal, and **0x13** is hexadecimal.

test The value to be compared with the value from the file. If the type is numeric, this value is specified in C form; if it is a string, it is specified as a C string with the usual escapes permitted (e.g. \n for new-line).

Numeric values may be preceded by a character indicating the operation to be performed. It may be =, to specify that the value from the file must equal the specified value, <, to specify that the value from the file must be less than the specified value, >, to specify that the value from the file must be greater than the specified value, or x to specify that any value will match. If the character is omitted, it is assumed to be =.

For string values, the byte string from the file must match the specified byte string; the byte string from the file which is matched is the same length as the specified byte string.

message The message to be printed if the comparison succeeds. If the string contains a *printf*(3S) format specification, the value from the file (with any specified masking performed) is printed using the message as the format string.

Some file formats contain additional information which is to be printed along with the file type. A line which begins with the character > indicates additional tests and messages to be printed. If the test on the line preceding the first line with a > succeeds, the tests specified in all the subsequent lines beginning with > are performed, and the messages printed if the tests succeed. The next line which does not begin with a > terminates this.

BUGS

There should be more than one level of subtests, with the level indicated by the number of > at the beginning of the line.

SEE ALSO

file(1)

NAME

mntent, fstab – static information about filesystems

SYNOPSIS

#include <mntent.h>

DESCRIPTION

The file */etc/fstab* describes the file systems and swapping partitions used by the local machine. It is created by the system administrator using a text editor and processed by commands which mount, unmount, check consistency of, dump and restore file systems, and by the system in providing swap space.

It consists of a number of lines of the form:

fsname dir type opts freq passno

an example of which would be:

/dev/xy0a / 4.2 rw,noquota 1 2

The entries in this file are accessed using the routines in *getmntent*(3), which returns a structure of the following form:

```
struct mntent {
        char    *mnt_fsname;    /* file system name */
        char    *mnt_dir;       /* file system path prefix */
        char    *mnt_type;      /* 4.2, nfs, swap, or ignore */
        char    *mnt_opts;      /* ro, quota, etc. */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
};
```

There is one entry per line in the file, and the fields are separated by white space. A '#' as the first non-white character indicates a comment.

The *mnt_opts* field consists of a string of comma seperated options. Some of the options are common to all filesystem types, others only make sense for a single filesystem type. See *mount*(8) for a more complete description of the options available.

The *mnt_type* field determines how the *mnt_fsname*, and *mnt_opts* fields will be interpreted. Below is a list of the file system types currently supported and the way each of them interprets these fields.

4.2

mnt_fsname	Must be a block special device.
mnt_opts	Valid opts are: ro, rw, suid, nosuid, quota, noquota.

NFS

mnt_fsname	The path on the server of the directory to be served.
mnt_opts	Valid opts are: ro, rw, suid, nosuid, hard, soft, bg, fg, retry, rsize, wsize, timeo, retrans, port, intr.

SWAP

mnt_fsname	Must be a block special device swap partition.
mnt_opts	Ignored.

If the *mnt_type* is specified as "ignore" the entry is ignored. This is useful to show disk partitions which are currently not used.

The field *mnt_freq* indicates how often each partition should be dumped by the *dump*(8) command (and triggers that commands w option which tells which file systems should be dumped). Most systems set the *mnt_freq* field to 1 indicating that the file systems are dumped each day.

The final field *mnt_passno* is used by the disk consistency check program *fsck*(8) to allow overlapped checking of file systems during a reboot. All file systems with *mnt_passno* of 1 are first checked simultaneosly, then all file systems with *mnt_passno* of 2, and so on. It is usual to make the *mnt_passno* of the root file system have the value 1 and then check one file system on each available disk drive in each subsequent pass to the exhaustion of file system partitions.

/etc/fstab is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because *fsck*, *mount*, and *umount* process the file sequentially; file systems must appear *after* file systems they are mounted within.

FILES

/etc/fstab

SEE ALSO

fsck(8), getmntent(3), mount(8), quotacheck(8), quotaon(8), umount(8)

NAME

mtab – mounted file system table

SYNOPSIS

/etc/mtab

#include <mntent.h>

DESCRIPTION

Mtab resides in the */etc* directory, and contains a table of filesystems currently mounted by the *mount* command. *Umount* removes entries from this file.

The file contains a line of information for each mounted filesystem, structurally identical to the contents of */etc/fstab*, described in *fstab*(5). There are a number of lines of the form:

fsname dir type opts freq passno

for example:

/dev/xy0a / 4.2 rw,noquota 1 2

The file is accessed by programs using *getmntent*(3), and by the system administrator using a text editor.

FILES

/etc/mtab

SEE ALSO

getmntent(3), fstab(5), mount(8)

NAME

netgroup – list of network groups

DESCRIPTION

Netgroup defines network wide groups, used for permission checking when doing remote mounts, remote logins, and remote shells. For remote mounts, the information in *netgroup* is used to classify machines; for remote logins and remote shells, it is used to classify users. Each line of the *netgroup* file defines a group and has the format

groupname member1 member2 ....

where member*i* is either another group name, or a triple:

(hostname, username, domainname)

Any of three fields can be empty, in which case it signifies a wild card. Thus

universal (,,)

defines a group to which everyone belongs. Field names that begin with something other than a letter, digit or underscore (such as "-") work in precisely the opposite fashion. For example, consider the following entries:

```
justmachines    (analytica,-,sun)
justpeople      (-,babbage,sun)
```

The machine *analytica* belongs to the group *justmachines* in the domain *sun*, but no users belong to it. Similarly, the user *babbage* belongs to the group *justpeople* in the domain *sun*, but no machines belong to it.

Network groups are contained in the yellow pages, and are accessed through these files:

/etc/yp/*domainname*/netgroup.dir
/etc/yp/*domainname*/netgroup.pag
/etc/yp/*domainname*/netgroup.byuser.dir
/etc/yp/*domainname*/netgroup.byuser.pag
/etc/yp/*domainname*/netgroup.byhost.dir
/etc/yp/*domainname*/netgroup.byhost.pag

These files can be created from */etc/netgroup* using *makedbm* (8).

FILES

/etc/netgroup
/etc/yp/*domainname*/netgroup.dir
/etc/yp/*domainname*/netgroup.pag
/etc/yp/*domainname*/netgroup.byuser.dir
/etc/yp/*domainname*/netgroup.byuser.pag
/etc/yp/*domainname*/netgroup.byhost.dir
/etc/yp/*domainname*/netgroup.byhost.pag

SEE ALSO

getnetgrent(3), exportfs(8), makedbm(8), ypserv(8)

NAME

netrc – .netrc file for ftp(1) remote login data

DESCRIPTION

The *.netrc* file contains data for logging in to a remote host over the network for file transfers by *ftp*(1). This file resides in the user's home directory on the machine initiating the file transfer. It's permissions should be set to disallow read access by group and others (see *chmod*(1V)).

Each line of the *.netrc* file defines options for a specific remote host. A line in the *.netrc* file can be either a **machine** line or a **default** line. The **default** line indicates a remote host to use as the default destination, and must be the first line in the *.netrc* file if present. The **machine** lines contain login information for each remote host to which files can be transferred.

default *default-machine-name*
machine *machine-name options*

Fields on each line are separated by SPACE or TAB characters.

The *options* for a **machine** line are:

Option	Parameter	Default	Description
login	name	localname	login name for remote machine
password	password	(none)	password for remote login name
command	command	(none)	default command to be executed
write	yes/no	yes	write to user if possible
force	yes/no	no	always prompt for login name and password
quiet	yes/no	no	llike the **-q** option

EXAMPLE

mahine ray login demo password mypassword

allows an autologin to the machine "ray" using the login name "demo" with password "mypassword".

FILES

~/.netrc

SEE ALSO

ftp(1), ftpd(8)

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A ''#'' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional ''.'' notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

passwd – password file

SYNOPSIS

/etc/passwd

DESCRIPTION

The *passwd* file contains for each user the following information:

name — User's login name — contains no upper case characters and must not be greater than eight characters long.

password — encrypted password

numerical user ID
: This is the user's ID in the system and it must be unique.

numerical group ID
: This is the number of the group that the user belongs to.

user's real name
: In some versions of UNIX, this field also contains the user's office, extension, home phone, and so on. For historical reasons this field is called the GCOS field.

initial working directory
: The directory that the user is positioned in when they log in — this is known as the 'home' directory.

shell — program to use as Shell when the user logs in.

The user's real name field may contain '&', meaning insert the login name.

The password file is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, */bin/sh* is used.

The *passwd* file can also have line beginning with a plus (+), which means to incorporate entries from the yellow pages. There are three styles of + entries: all by itself, + means to insert the entire contents of the yellow pages password file at that point; *+name* means to insert the entry (if any) for *name* from the yellow pages at that point; *+@name* means to insert the entries for all members of the network group *name* at that point. If a + entry has a non-null password, directory, gecos, or shell field, they will overide what is contained in the yellow pages. The numerical user ID and group ID fields cannot be overridden.

EXAMPLE

Here is a sample */etc/passwd* file:

```
root:q.mJzTnu8icF.:0:10:God:/:/bin/csh
tut:6k/7KCFRPNVXg:508:10:Bill Tuthill:/usr2/tut:/bin/csh
+john:
+@documentation:no-login:
+:::Guest
```

In this example, there are specific entries for users *root tut*, in case the yellow pages are out of order. The user will have his password entry in the yellow pages incorporated without change; anyone in the netgroup *documentation* will have their password field disabled, and anyone else will be able to log in with their usual password, shell, and home directory, but with a gecos field of *Guest.*

The password file resides in the */etc* directory. Because of the encrypted passwords, it has general read permission and can be used, for example, to map numerical user ID's to names.

Appropriate precautions must be taken to lock the */etc/passwd* file against simultaneous changes if it is to be edited with a text editor; *vipw*(8) does the necessary locking.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5), vipw(8), adduser(8)

NAME

phones – remote host phone number data base

SYNOPSIS

/etc/phones

DESCRIPTION

The file */etc/phones* contains the system-wide private phone numbers for the *tip*(1C) program. */etc/phones* is normally unreadable, and so may contain privileged information. The format of */etc/phones* is a series of lines of the form: <system-name>[\t]*<phone-number>. The system name is one of those defined in the *remote*(5) file and the phone number is constructed from [0123456789–=*%]. The '=' and '*' characters are indicators to the auto call units to pause and wait for a second dial tone (when going through an exchange). The '=' is required by the DF02-AC and the '*' is required by the BIZCOMP 1030.

Comment lines are lines containing a '#' sign in the first column of the line.

Only one phone number per line is permitted. However, if more than one line in the file contains the same system name *tip*(1C) will attempt to dial each one in turn, until it establishes a connection.

FILES

/etc/phones

SEE ALSO

tip(1C), remote(5)

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X), and are interpreted for various devices by commands described in *plot*(1G). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l, m, n,** or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See *plot*(1G).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver*.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

4014	space(0, 0, 3120, 3120);
ver	space(0, 0, 2048, 2048);
300, 300s	space(0, 0, 4096, 4096);
450	space(0, 0, 4096, 4096);

SEE ALSO

plot(1G), plot(3X), graph(1G)

NAME

printcap – printer capability data base

SYNOPSIS

/etc/printcap

DESCRIPTION

Printcap is a simplified version of the *termcap*(5) data base for describing printers. The spooling system accesses the *printcap* file every time it is used, allowing dynamic addition and deletion of printers. Each entry in the data base describes one printer. This data base may not be substituted for, as is possible for *termcap*, because it may allow accounting to be bypassed.

The default printer is normally *lp*, though the environment variable *PRINTER* may be used to override this. Each spooling utility supports a **–P***printer* option to explicitly name a destination printer.

Refer to the *Line Printer Spooler Manual* in the Sun *System Administration Manual* for a discussion of how to set up the database for a given printer.

Each entry in the *printcap* file describes a printer, and is a line consisting of a number of fields separated by ':' characters. The first entry for each printer gives the names which are known for the printer, separated by '|' characters. The first name is conventionally a number. The second name given is the most common abbreviation for the printer, and the last name given should be a long name fully identifying the printer. The second name should contain no blanks; the last name may well contain blanks for readability. Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability.

Capabilities in *printcap* are all introduced by two-character codes, and are of three types:

Boolean capabilities indicate that the printer has some particular feature. Boolean capabilities are simply written between the ':' characters, and are indicated by the word 'bool' in the **type** column of the capabilities table below.

Numeric capabilities supply information such as baud-rates, number of lines per page, and so on. Numeric capabilities are indicated by the word 'num' in the **type** column of the capabilities table below. Numeric capabilities are given by the two-character capability code followed by the '#' character, followed by the numeric value. For example: :br#1200: is a numeric entry stating that this printer should run at 1200 baud.

String capabilities give a sequence which can be used to perform particular printer operations such as cursor motion. String valued capabilities are indicated by the word 'str' in the **type** column of the capabilities table below. String valued capabilities are given by the two-character capability code followed by an '=' sign and then a string ending at the next following ':'. For example, :rp=spinwriter: is a sample entry stating that the remote printer is named 'spinwriter'.

CAPABILITIES

Name	Type	Default	Description
af	str	NULL	name of accounting file
br	num	none	if lp is a tty, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	TeX data filter (DVI format)
du	str	0	User ID of user 'daemon'.
fc	num	0	if lp is a tty, clear flag bits (sgtty.h)
ff	str	"\f"	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like 'fc' but set bits
gf	str	NULL	graph data filter (*plot*(3X) format)
ic	bool	false	driver supports (non standard) ioctl call for indenting printout
if	str	NULL	name of text filter which does accounting

lf	str	"/dev/console"	error logging file name
lo	str	"lock"	name of lock file
lp	str	"/dev/lp"	device name to open for output
mc	num	0	maximum number of copies
mx	num	1000	maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	ditroff data filter (device independent troff)
of	str	NULL	name of output filtering program
pl	num	66	page length (in lines)
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	"lp"	remote printer name argument
rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open printer device read/write instead of read-only
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	"/usr/spool/lpd"	spool directory
sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	"status"	status file name
tf	str	NULL	troff data filter (cat phototypesetter)
tr	str	NULL	trailer string to print when queue empties
vf	str	NULL	raster image filter
xc	num	0	if lp is a tty, clear local mode bits (tty (4))
xs	num	0	like 'xc' but set bits

Error messages sent to the console have a carriage return and a line feed appended to them, rather than just a line feed.

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

Note that the 'fs', 'fc', 'xs', and 'xc' fields are flag *masks* rather than flag *values*. Certain default device flags are set when the device is opened by the lineprinter daemon if the device is a tty. The flags indicated in the 'fc' field are then cleared; the flags in the 'fs' field are then set (or vice-versa, depending on the order of 'fc#nnnn' and 'fs#nnnn' in the /etc/printcap file). For example, to set exactly the flags 06300 in the 'fs' field, do:

```
:fc#0177777:fs#06300:
```

The same process applies to the 'xc' and 'xs' fields.

SEE ALSO

termcap(5), lpc(8), lpd(8), pac(8), lpr(1), lpq(1), lprm(1)
The *Line Printer Spooler Manual* in the Sun *System Administration Manual*.

NAME

protocols – protocol name data base

SYNOPSIS

/etc/protocols

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

The following example is taken from the Sun UNIX system.

```
#
# Internet (IP) protocols
#
ip          0       IP      # internet protocol, pseudo protocol number
icmp        1       ICMP    # internet control message protocol
ggp         2       GGP     # gateway-gateway protocol
tcp         6       TCP     # transmission control protocol
pup         12      PUP     # PARC universal packet protocol
udp         17      UDP     # user datagram protocol
```

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

rasterfile – Sun's file format for raster images

SYNOPSIS

#include <rasterfile.h>

DESCRIPTION

A rasterfile is composed of three parts: first, a header containing 8 integers; second, a (possibly empty) set of colormap values; and third, the pixel image, stored a line at a time, in increasing *y* order. The image is layed out in the file as in a memory pixrect. Each line of the image is rounded up to the nearest 16 bits.

The header is defined by the following structure:

```
struct rasterfile {
        int     ras_magic;
        int     ras_width;
        int     ras_height;
        int     ras_depth;
        int     ras_length;
        int     ras_type;
        int     ras_maptype;
        int     ras_maplength;
};
```

The *ras_magic* field always contains the following constant:

```
#define RAS_MAGIC    0x59a66a95
```

The *ras_width*, *ras_height*, and *ras_depth* fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is either 1 or 8, corresponding to standard frame buffer depths. The *ras_length* field contains the length in bytes of the image data. For an unencoded image, this number is computable from the *ras_width*, *ras_height*, and *ras_depth* fields, but for an encoded image it must be explicitly stored in order to be available without decoding the image itself. Note that the length of the header and of the (possibly empty) colormap values are not included in the value of the *ras_length* field; it is only the image data length. For historical reasons, files of type *RT_OLD* will usually have a 0 in the *ras_length* field, and software expecting to encounter such files should be prepared to compute the actual image data length if needed. The *ras_maptype* and *ras_maplength* fields contain the type and length in bytes of the colormap values, respectively. If *ras_maptype* is not *RMT_NONE* and the *ras_maplength* is not 0, then the colormap values are the *ras_maplength* bytes immediately after the header. These values are either uninterpreted bytes (usually with the *ras_maptype* set to *RMT_RAW*) or the equal length red, green and blue vectors, in that order (when the *ras_maptype* is *RMT_EQUAL_RGB*). In the latter case, the *ras_maplength* must be three times the size in bytes of any one of the vectors.

FILES

/usr/include/rasterfile.h

SEE ALSO

Programmer's Reference Manual for SunWindows

NAME

remote – remote host description file

SYNOPSIS

/etc/remote

DESCRIPTION

The systems known by *tip*(1C) and their attributes are stored in an ASCII file which is structured somewhat like the *termcap*(5) file. Each line in the file provides a description for a single *system*. Fields are separated by a colon (':'). Lines ending in a \ character with an immediately following newline are continued on the next line.

The first entry is the name(s) of the host system. If there is more than one name for a system, the names are separated by vertical bars. After the name of the system comes the fields of the description. A field name followed by an '=' sign indicates a string value follows. A field name followed by a '#' sign indicates a following numeric value.

Entries named 'tip*' and 'cu*' are used as default entries by *tip*, and the *cu* interface to *tip*, as follows. When *tip* is invoked with only a phone number, it looks for an entry of the form 'tip300', where 300 is the baud rate with which the connection is to be made. When the *cu* interface is used, entries of the form 'cu300' are used.

CAPABILITIES

Capabilities are either strings (str), numbers (num), or boolean flags (bool). A string capability is specified by *capability=value*; for example, 'dv=/dev/harris'. A numeric capability is specified by *capability#value*; for example, 'xa#99'. A boolean capability is specified by simply listing the capability.

at (str) Auto call unit type.

br (num) The baud rate used in establishing a connection to the remote host. This is a decimal number. The default baud rate is 300 baud.

cm (str) An initial connection message to be sent to the remote host. For example, if a host is reached through port selector, this might be set to the appropriate sequence required to switch to the host.

cu (str) Call unit if making a phone call. Default is the same as the 'dv' field.

di (str) Disconnect message sent to the host when a disconnect is requested by the user.

du (bool) This host is on a dial-up line.

dv (str) UNIX device(s) to open to establish a connection. If this file refers to a terminal line, *tip*(1C) attempts to perform an exclusive open on the device to insure only one user at a time has access to the port.

el (str) Characters marking an end-of-line. The default is NULL. *Tip* only recognizes '~' escapes after one of the characters in 'el', or after a carriage-return.

fs (str) Frame size for transfers. The default frame size is equal to BUFSIZ.

hd (bool) The host uses half-duplex communication, local echo should be performed.

ie (str) Input end-of-file marks. The default is NULL.

oe (str) Output end-of-file string. The default is NULL. When *tip* is transferring a file, this string is sent at end-of-file.

pa (str) The type of parity to use when sending data to the host. This may be one of 'even', 'odd', 'none', 'zero' (always set bit 8 to zero), 'one' (always set bit 8 to 1). The default is 'none'.

pn (str) Telephone number(s) for this host. If the telephone number field contains an @ sign, *tip* searches the */etc/phones* file for a list of telephone numbers — see *phones*(5). A % sign in the telephone number indicates a 5-second delay for the Ventel Modem.

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

Here is a short example showing the use of the capability continuation feature:

```
UNIX-1200:\
        :dv=/dev/cua0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
        :pn=7654321%:tc=UNIX-1200
```

FILES

/etc/remote

SEE ALSO

tip(1C), phones(5)

NAME

resolver – configuration file for name server routines

DESCRIPTION

The resolver configuration file contains information that is read by the resolver routines the first time they are invoked in a process. The file is designed to be human readable and contains a list of name-value pairs that provide various types of resolver information.

The different configuration options are:

nameserver
followed by the Internet address (in dot notation) of a name server that the resolver should query. At least one name server should be listed. Up to MAXNS (currently 3) name servers may be listed, in that case the resolver library queries tries them in the order listed. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made).

domain followed by a domain name, that is the default domain to append to names that do not have a dot in them. This defaults to the domain set by the domainname(1) command.

address followed by an Internet address (in dot notation) of any preferred networks. The list of addresses returned by the resolver will be sorted to put any addresses on this network before any others.

The name value pair must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

FILES

/etc/resolv.conf

SEE ALSO

domainname(1), gethostent(3N), named(8C)

NAME

rmtab – local file system mount statistics

DESCRIPTION

Rmtab resides in directory */etc* and contains a record of all clients that have done remote mounts of file systems from this machine. Whenever a remote *mount* is done, an entry is made in the *rmtab* file of the machine serving up that file system. *Umount* removes entries, if of a remotely mounted file system. *Umount –a* broadcasts to all servers, and informs them that they should remove all entries from *rmtab* created by the sender of the broadcast message. By placing a *umount –a* command in */etc/rc.boot*, *rmtab* tables can be purged of entries made by a crashed host, which upon rebooting did not remount the same file systems it had before. The table is a series of lines of the form

hostname:directory

This table is used only to preserve information between crashes, and is read only by *mountd*(8) when it starts up. *Mountd* keeps an in-core table, which it uses to handle requests from programs like *showmount*(1) and *shutdown*(8).

FILES

/etc/rmtab

SEE ALSO

showmount(1), mountd(8), mount(8), umount(8), shutdown(8)

BUGS

Although the *rmtab* table is close to the truth, it is not always 100% accurate.

NAME

rpc – rpc program number data base

SYNOPSIS

/etc/rpc

DESCRIPTION

The *rpc* file contains user readable names that can be used in place of rpc program numbers. Each line has the following information:

name of server for the rpc program
rpc program number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Here is an example of the */etc/rpc* file from the Sun UNIX System.

```
#
#              rpc  1.7  86/04/24
#
portmapper      100000          portmap sunrpc
rstatd                          100001          rstat rup perfmeter
rusersd                         100002          rusers
nfs                             100003          nfsprog
ypserv                          100004          ypprog
mountd                          100005          mount showmount
ypbind                          100007
walld                           100008          rwall shutdown
yppasswdd       100009          yppasswd
etherstatd      100010          etherstat
rquotad                         100011          rquotaprog quota rquota
sprayd                          100012          spray
3270_mapper     100013
rje_mapper      100014
selection_svc   100015          selnsvc
database_svc    100016
rexd                            100017          rex
alis                            100018
sched                           100019
llockmgr        100020
nlockmgr        100021
x25.inr                         100022
statmon                         100023
status                          100024
```

FILES

/etc/rpc

SEE ALSO

getrpcent(3N)

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> **DDDDD DDDDD**
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
.
.
.
@c <comments> ...
.
.
.
@e

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one MR number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>
@f v <program name>
@f i
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>
@f n
@f j
@f l <lock-releases>
@f q <user defined>

The **t** flag defines the replacement for the identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **–b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the sccsfile.5 identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (for example, when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **–e** keyletter). The **q** flag defines the replacement for the identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines **@t** and **@T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

NAME

servers – inet server data base

DESCRIPTION

The *servers* file contains the list of servers that *inetd*(8) operates. For each server a single line should be present with the following information:

name of server
protocol
server location

If the server is rpc based, then the name field should be **rpc**, and following the server location are two additional fields, one with the rpc program number, the second with either a version number or a range of version numbers.

Items are separated by any number of blanks and/or tab characters. A ''#'' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

The name of the server should be the official service name as contained in *services*(5). The protocol entry is either udp or tcp. The server location is the full path name of the server program.

EXAMPLE

The following example is taken from the Sun UNIX system.

```
tcp       tcp   /usr/etc/in.tcpd
telnet    tcp   /usr/etc/in.telnetd
shell     tcp   /etc/in.rshd
login     tcp   /etc/in.rlogind
exec      tcp   /usr/etc/in.rexecd
tcp       udp   /usr/etc/in.ttcpd
syslog    udp   /usr/etc/in.syslog
comsat    udp   /usr/etc/in.comsat
talk      udp   /usr/etc/in.talkd
time      tcp   /usr/etc/in.timed
rpc       udp   /usr/etc/rpc.rstatd    100001  1-2
rpc       udp   /usr/etc/rpc.rusersd   100002  1
rpc       udp   /usr/etc/rpc.rwalld    100008  1
rpc       udp   /usr/etc/rpc.mountd    100005  1
```

FILES

/etc/servers

SEE ALSO

services(5), inetd(8)

BUGS

Because of a limitation on the number of open files, this file must contain fewer than 27 lines.

NAME

services – service name data base

SYNOPSIS

/etc/services

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (for instance, "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

EXAMPLE

Here is an example of the */etc/services* file from the Sun UNIX System.

```
#
# @(#)services 1.7 86/02/28 SMI
#
# Network services, Internet style
# This file is never consulted when the yellow pages are running
#
echo                    7/udp
discard                 9/udp                   sink null
systat                  11/tcp
daytime                 13/tcp
netstat                 15/tcp
ftp-data        20/tcp
ftp                     21/tcp
telnet                  23/tcp
smtp                    25/tcp                  mail
time                    37/tcp                  timserver
time                    37/udp                  timserver
name                    42/udp                  nameserver
whois                   43/tcp                  nicname# usually to sri-nic
domain                  53/udp
domain                  53/tcp
hostnames       101/tcp                 hostname        # usually to sri-nic
sunrpc                  111/udp
sunrpc                  111/tcp
#
# Host specific functions
#
tftp                    69/udp
rje                     77/tcp
finger                  79/tcp
link                    87/tcp                  ttylink
supdup                  95/tcp
```

```
csnet-ns        105/tcp
uucp-path       117/tcp
untp                            119/tcp                 usenet
ntp                             123/tcp
ingreslock      1524/tcp
#
# UNIX specific services
#
exec                            512/tcp
login                           513/tcp
shell                           514/tcp                 cmd# no passwords used
printer                         515/tcp                 spooler# experimental
courier                         530/tcp                 rpc# experimental
biff                            512/udp                 comsat
who                             513/udp                 whod
syslog                          514/udp
talk                            517/udp
route                           520/udp                 router routed
new-rwho        550/udp                         new-who # experimental
rmonitor        560/udp                         rmonitord # experimental
monitor                         561/udp                 # experimental
```

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

statmon, current, backup, state – statd directory and file structures

SYNOPSIS

/etc/statmon/current /etc/statmon/backup, /etc/statmon/state

DESCRIPTION

/etc/statmon/current and */etc/statmon/backup* are directories generated by *statd*. Each entry in */etc/statmon/current* represents the name of the machine to be monitored by the *statd* daemon. Each entry in */etc/statmon/backup* represents the name of the machine to be notified by the *statd* daemon upon its recovery.

/etc/statmon/state is a file generated by *statd* to record its version number. This version number is incremented each time a crash or recovery takes place.

SEE ALSO

statd(8C), lockd(8C)

NAME

tar – tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar*(1) command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads, unless the **B** keyletter is used.

The header block looks like:

```
#define TBLOCK          512
#define NAMSIZ 100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII '0' if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The name linked-to, if any, is in *linkname*, with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually re-scanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

term – terminal driving tables for nroff

SYNOPSIS

/usr/lib/term/tabname

DESCRIPTION

Nroff(1) uses driving tables to customize its output for various types of output devices, such as terminals, line printers, daisy-wheel printers, or special output filter programs. These driving tables are written as C programs, compiled, and installed in the directory */usr/lib/term*. The *name* of the output device is specified with the –T option of *nroff*. The structure of the terminal table is as follows:

```
#define     INCH      240

struct {
            int bset;
            int breset;
            int Hor;
            int Vert;
            int Newline;
            int Char;
            int Em;
            int Halfline;
            int Adj;
            char *twinit;
            char *twrest;
            char *twnl;
            char *hlr;
            char *hlf;
            char *flr;
            char *bdon;
            char *bdoff;
            char *ploton;
            char *plotoff;
            char *up;
            char *down;
            char *right;
            char *left;
            char *codetab[256–32];
            char *zzz;
} t;
```

The meanings of the various fields are as follows:

bset bits to set in the *sg_flags* field of the *sgtty* structure before output; see *tty*(4).

breset bits to reset in the *sg_flags* field of the *sgtty* structure after output; see *tty*(4).

Hor horizontal resolution in fractions of an inch.

Vert vertical resolution in fractions of an inch.

Newline space moved by a newline (linefeed) character in fractions of an inch.

Char quantum of character sizes, in fractions of an inch. (that is, a character is a multiple of Char units wide)

Em size of an em in fractions of an inch.

Halfline space moved by a half-linefeed (or half-reverse-linefeed) character in fractions of an inch.

Adj quantum of white space, in fractions of an inch. (that is, white spaces are a multiple of Adj units wide)

Note: if this is less than the size of the space character (in units of Char; see below for how the sizes of characters are defined), *nroff* will output fractional spaces using plot mode. Also, if the –e switch to *nroff* is used, Adj is set equal to Hor by *nroff*.

twinit set of characters used to initialize the terminal in a mode suitable for *nroff*.

twrest set of characters used to restore the terminal to normal mode.

twnl set of characters used to move down one line.

hlr set of characters used to move up one-half line.

hlf set of characters used to move down one-half line.

flr set of characters used to move up one line.

bdon set of characters used to turn on hardware boldface mode, if any.

bdoff set of characters used to turn off hardware boldface mode, if any.

ploton set of characters used to turn on hardware plot mode (for Diablo type mechanisms), if any.

plotoff set of characters used to turn off hardware plot mode (for Diablo type mechanisms), if any.

up set of characters used to move up one resolution unit (Vert) in plot mode, if any.

down set of characters used to move down one resolution unit (Vert) in plot mode, if any.

right set of characters used to move right one resolution unit (Hor) in plot mode, if any.

left set of characters used to move left one resolution unit (Hor) in plot mode, if any.

codetab definition of characters needed to print an *nroff* character on the terminal. The first byte is the number of character units (Char) needed to hold the character; that is, "\001" is one unit wide, "\002" is two units wide, etc. The high-order bit (0200) is on if the character is to be underlined in underline mode (.ul). The rest of the bytes are the characters used to produce the character in question. If the character has the sign (0200) bit on, it is a code to move the terminal in plot mode. It is encoded as:

0100 bit on	vertical motion.
0100 bit off	horizontal motion.
040 bit on	negative (up or left) motion.
040 bit off	positive (down or right) motion.
037 bits	number of such motions to make.

zzz a zero terminator at the end.

All quantities which are in units of fractions of an inch should be expressed as INCH**num/denom*, where *num* and *denom* are respectively the numerator and denominator of the fraction; that is, 1/48 of an inch would be written as "INCH/48".

If any sequence of characters does not pertain to the output device, that sequence should be given as a null string.

FILES

*/usr/lib/term/tab*name	driving tables
/usr/lib/term/README	list of terminals supported by *nroff*(1)

SEE ALSO

nroff(1)

NAME

term – format of compiled term file

SYNOPSIS

term

DESCRIPTION

Compiled *terminfo* descriptions are placed under the directory **/usr/5lib/terminfo**. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: */usr/5lib/terminfo/c/name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, *act4* can be found in the file **/usr/5lib/terminfo/a/act4**. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8 or more bit byte is assumed, but no assumptions about byte ordering or sign extension are made.

The compiled file is created with the *tic* program, and read by the routine *setupterm*. Both of these pieces of software are part of *curses*(3V). The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal 0432); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value –1 is represented by 0377, 0377, other negative value are illegal. The –1 generally means that a capability is missing from this terminal. Note that this format corresponds to the hardware of the VAX and PDP-11. Machines where this does not correspond to the hardware read the integers as two bytes and compute the result.

The terminal names section comes next. It contains the first line of the terminfo description, listing the various names for the terminal, separated by the '|' character. The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either 0 or 1 as the flag is present or absent. The capabilities are in the same order as the file **<term.h>**.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is –1, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of –1 means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information $<nn> and parameter information %x are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for *setupterm* to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since *setupterm* has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine *setupterm* must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the Microterm ACT 4 is included:

```
microterm|act4|microterm act iv,
   cr=^M, cud1=^J, ind=^J, bel=^G, am, cub1=^H,
   ed=^_, el=^^, clear=^L, cup=^T%p1%c%p2%c,
   cols#80, lines#24, cuf1=^X, cuu1=^Z, home=^],
```

```
000 032 001      \0 025  \0  \b  \0 212  \0   "  \0   m   i   c   r
020   o   t   e   r   m   |   a   c   t   4   |   m   i   c   r   o
040   t   e   r   m       a   c   t       i   v  \0  \0 001  \0  \0
060  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
100  \0  \0   P  \0 377 377 030  \0 377 377 377 377 377 377 377 377
120 377 377 377 377  \0  \0 002  \0 377 377 377 377 004  \0 006  \0
140  \b  \0 377 377 377 377  \n  \0 026  \0 030  \0 377 377 032  \0
160 377 377 377 377 034  \0 377 377 036  \0 377 377 377 377 377 377
200 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
520 377 377 377 377      \0 377 377 377 377 377 377 377 377 377 377
540 377 377 377 377 377 377 007  \0  \r  \0  \f  \0 036  \0 037  \0
560 024   %   p   1   %   c   %   p   2   %   c  \0  \n  \0 035  \0
600  \b  \0 030  \0 032  \0  \n  \0
```

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/5lib/terminfo/*/* compiled terminal capability data base

SEE ALSO

curses(3V), terminfo(5V).

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, for example, by *vi*(1) and *curses*(3X). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Each entry in the *termcap* file describes a terminal, and is a line consisting of a number of fields separated by ':' characters. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability. Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability.

Capabilities in *termcap* are all introduced by two-character codes, and are of three types:

Boolean capabilities indicate that the terminal has some particular feature. Boolean capabilities are simply written between the ':' characters, and are indicated by the word 'bool' in the **type** column of the capabilities table below.

Numeric capabilities supply information such as the size of the terminal or the size of particular delays. Numeric capabilities are indicated by the word 'num' in the **type** column of the capabilities table below. Numeric capabilities are given by the two-character capability code followed by the '#' character and then the numeric value. For example: :co#80: is a numeric entry stating that this terminal has 80 columns.

String capabilities give a sequence which can be used to perform particular terminal operations such as cursor motion. String valued capabilities are indicated by the word 'str' in the **type** column of the capabilities table below. String valued capabilities are given by the two-character capability code followed by an '=' sign and then a string ending at the next following ':'. For example, :ce=16\E^S: is a sample entry for clear to end-of-line.

CAPABILITIES

(P) indicates padding may be specified

(P*) indicates that padding may be based on the number of lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bl	str		Audible bell character
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)

cs	str	(P)	Change scrolling region (vt100), like cm
ct	str		Clear all tab stops
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give ":ei=:" if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give ":im=:" if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
le	str		Move cursor left one place
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
mb	str		Turn on blinking
md	str		Enter bold (extra-bright) mode
me	str		Turn off all attributes, normal mode
mh	str		Enter dim (half-bright) mode
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mr	str		Enter reverse mode
ms	bool		Safe to move while in standout and underline mode

mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
rf	str		Reset file, like **if** but for *reset* (1)
rs	str		Reset string, like **is** but for *reset* (1)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by **so** or **se**
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
st	str		Set a tab in all rows, current column
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by **us** or **ue**
ul	bool		Terminal underlines even though it doesn't overstrike
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
vt	num		Virtual terminal number (CB/UNIX)
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like **ce** \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following example describes the wyse terminal entry.

```
wv|wyse-vp|wyse|Wyse 50 in ADDS Viewpoint emulation mode with "enhance" on:
:am:cr=^M:do=^J:nl=^J:bl=^G:if=/usr/lib/tabset/wyse-adds: :le=^H:bs:li#24:co#80:cm=EY%+ %+
:cd=Ek:ce=EK:nd=^F: :up=^Z:cl=^L:ll=^A:kl=^U:kr=^F:kd=^J:ku=^Z:kh=^A:
:pt:so=^N:se=^O:us=^N:ue=^O: :dl=El:al=EM:im=Eq:ei=Er:dc=EW:
:is=E' 72E'9^OEr:rs=E' 72E'9^OEr:
```

Types of Capabilities

Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations. All capabilities have two letter codes.

Boolean capabilities are introduced simply by stating the two-character capability code in the field between ':' characters. For instance, the fact that the Concept has "automatic margins" (that is, an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**.

Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

String valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, for instance, '20', or an integer followed by an '*', that is, '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late so that a \200 comes out as a \000 would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *termcap* and to build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in */etc/termcap*. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor.

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than **^H** (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the *termcap* file usually assumes that this is on, that is, **am**.

These capabilities suffice to describe hardcopy and ''glass-tty'' terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with *printf*(3S) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is "cm=6\E&%r%2c%2Y". The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cm=^T%.%.". Terminals which use "%." need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit \t, \n ^D and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ ".

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *termcap*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (for example, if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **sg** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If they leave blank spaces on the screen, set **ug**. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, for example, from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

ANSI terminals have modes for the character highlighting. Dim characters may be generated in dim mode, entered by **mh**; reverse video characters in reverse mode, entered by **mr**; bold characters in bold mode, entered by **md**; and normal mode characters restored by turning off all attributes with **me**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, ":ko=cl,ll,sf,sb:", which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than **^I** to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow "~" characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities

given at the left override the ones in the similar terminal. A capability can be canceled with **xx@** where xx is the capability. For example, the entry

hn|2621nl:ks@:ke@:tc=2621:

defines a **2621nl** that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3X), termcap(3X), tset(1), vi(1), ul(1), more(1)

BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap*(3X) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/5lib/terminfo/*/*

DESCRIPTION

terminfo is a data base describing terminals, used by *curses*(3V). Terminals are described in *terminfo* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *terminfo*.

Entries in *terminfo* consist of a number of ',' separated fields. White space after each ',' is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a VT100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	c100-na
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The cap-name is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two letter internal code used in the compiled database, and always corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file caps to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

(P) indicates that padding may be specified

(G) indicates that the string is passed through *tparm* with parms as given (#*i*).

(*) indicates that padding may be based on the number of lines affected

($\#_i$) indicates the i^{th} parameter.

Variable Booleans	Cap-name	I. Code	Description
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)

eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g., dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print ~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (UNIX system)
width_status_line,	wsl	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (VT100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cud1	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)

cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)
delete_line,	dl1	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	smso	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	il1	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key

key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	ka	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kil1	kA	Sent by insert line
key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcuf1	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuu1	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2

pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2
print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_1string,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept 100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100|c100|concept|c104|c100-4p|concept 100,
    am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
    cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
    cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
    ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
    il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev    $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
    rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
    smcup=\EU\Ev  8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
    smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence

which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the two-character code, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both \E and \e map to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences \n \l \r \t \b \f \s give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for ^, \\ for \, \, for comma, \: for :, and \0 for null. (\0 will produce \200, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a \.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with some *curses*-based application to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *terminfo* file to describe it or bugs in the application. To easily test a new terminal description you can set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in **/usr/5lib/terminfo**. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to insert 16 lines into the middle of a full screen at 9600 baud. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over, for example, you would not normally use '**cuf1**= ' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
33|tty33|tty|model 33 teletype,
bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM–3 is described as

```
adm3|3|lsi adm3,
am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with *printf*(3S) like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special % codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The % encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	
%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c

%{nn}	integer constant nn
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)

%? expr %t thenpart %e elsepart %;
if-then-else, %e elsepart is optional.
else-if's are possible ala Algol 68:
%? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e %;
c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the HP 2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is "cup=6\E&%p2%2dc%p1%2dY".

The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cup=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit \n ^D and \r, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cup=\E=%p1%' '%+%c%p2%' '%+%c". After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup** . If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the Tektronix 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on HP terminals cannot be used for **home**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **Ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command – the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half-bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smacs** (enter alternate character set mode) and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by sgr, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by terminfo.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10**. If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1, is2,** and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1, rs2, rf,** and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the VT100 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr, ind, cub1, ff,** and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit H19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, tparm(repeat_char, 'x', 10) is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a CC variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The

difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches and Braindamage

Hazeltine terminals, which do not allow '~' characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and VT100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where xx is the capability. For example, the entry

2621-nl, smkx@, rmkx@, use=2621,

defines a 2621-nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/usr/5lib/terminfo/?/* files containing terminal descriptions

SEE ALSO

curses(3V), printf(3S)

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/5lib/terminfo/*/*

DESCRIPTION

terminfo is a data base describing terminals, used by *curses*(3V). Terminals are described in *terminfo* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *terminfo*.

Entries in *terminfo* consist of a number of ',' separated fields. White space after each ',' is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a VT100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-*n*	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	c100-na
-*n*p	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The cap-name is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two letter internal code used in the compiled database, and always corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file **caps** to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

(P) indicates that padding may be specified

(G) indicates that the string is passed through *tparm* with parms as given (#*i*).

(*) indicates that padding may be based on the number of lines affected

($\#_i$) indicates the i^{th} parameter.

Variable Booleans	Cap-name	I. Code	Description
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)

eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g., dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print ~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (UNIX system)
width_status_line,	wsl	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (VT100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cud1	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)

cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)
delete_line,	dl1	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	smso	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	il1	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (p*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key

key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	ka	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kil1	kA	Sent by insert line
key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcuf1	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuu1	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2

pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2
print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_1string,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1
underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept 100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100|c100|concept|c104|c100-4p|concept 100,
    am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
    cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
    cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
    ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
    il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev    $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
    rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
    smcup=\EU\Ev  8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
    smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence

which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the two-character code, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences **\n \l \r \t \b \f \s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include \^ for ^, \\ for \, \, for comma, \: for :, and \0 for null. (\0 will produce \200, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a \.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with some *curses*-based application to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *terminfo* file to describe it or bugs in the application. To easily test a new terminal description you can set the environment variable TERMINFO to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in **/usr/5lib/terminfo**. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to insert 16 lines into the middle of a full screen at 9600 baud. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (os applies to storage scope terminals, such as Tektronix 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over, for example, you would not normally use '**cuf1**= ' because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
33|tty33|tty|model 33 teletype,
bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM–3 is described as

```
adm3|3|lsi adm3,
am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with *printf*(3S) like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The **%** encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	
%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c

%{nn}	integer constant nn
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)

%? expr %t thenpart %e elsepart %;
if-then-else, %e elsepart is optional.
else-if's are possible ala Algol 68:
%? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e %;
c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the HP 2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is "cup=6\E&%p2%2dc%p1%2dY".

The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cup=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit \n ^D and \r, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cup=\E=%p1%' '%+%c%p2%' '%+%c". After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the Tektronix 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on HP terminals cannot be used for **home**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **Ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command – the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half-bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smacs** (enter alternate character set mode) and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by terminfo.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10**. If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1, is2,** and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1, rs2, rf,** and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the VT100 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr, ind, cub1, ff,** and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit H19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, tparm(repeat_char, 'x', 10) is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a CC variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The

difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches and Braindamage

Hazeltine terminals, which do not allow '~' characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and VT100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a "magic cookie", that to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsb**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where xx is the capability. For example, the entry

2621-nl, smkx@, rmkx@, use=2621,

defines a 2621-nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/usr/5lib/terminfo/?/* files containing terminal descriptions

SEE ALSO

curses(3V), printf(3S)

NAME

tp – DEC/mag tape formats

DESCRIPTION

Tp dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See *reboot*(8).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct {
        char            pathname[32];
        unsigned short  mode;
        char            uid;
        char            gid;
        char            unused1;
        char            size[3];
        long            modtime;
        unsigned short  tapeaddr;
        char            unused2[16];
        unsigned short  checksum;
};
```

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (see file system *fs*(5)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks above 25 (resp. 63) are available for file storage.

A fake entry has a size of zero.

SEE ALSO

fs(5)

BUGS

The *pathname*, *uid*, *gid*, and *size* fields are too small.

NAME

ttys – terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them so that people can log in. There is one line in the *ttys* file per special file associated with a terminal.

The first character of a line in the *ttys* file is either '0' or '1'. If the first character on the line is a '0', the *init* program ignores that line. If the first character on the line is a '1', the *init* program creates a login process for that line.

The second character on each line is used as an argument to *getty*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the second character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. The remainder of the line is the terminal's entry in the device directory, /dev.

Getty uses the second character in the *ttys* file to look up the characteristics of the terminal in the */etc/gettytab* file. Consult the *gettytab*(5) manual page for an explanation of the layout of */etc/gettytab*.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1), gettytab(5)

NAME

ttytype – data base of terminal types by port

SYNOPSIS

/etc/ttytype

DESCRIPTION

Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name listed in termcap (5)), a space, and the name of the tty, minus /dev/.

This information is read by *tset*(1) and by *login*(1) to initialize the TERM variable at login time.

SEE ALSO

tset(1), login(1)

BUGS

Some lines are merely known as ''dialup'' or ''plugboard''.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
/*      @(#)types.h 1.2 86/08/01 SMI; from UCB 4.11 83/07/01*/

/*
 * Basic system types and major/minor device constructing/busting macros.
 */
#ifndef _TYPES_
#define _TYPES_

#ifndef KERNEL
#include<sys/sysmacros.h>
#else
#include"../h/sysmacros.h"
#endif

typedef  unsigned char   u_char;
typedef  unsigned short  u_short;
typedef  unsigned int    u_int;
typedef  unsigned long   u_long;
typedef  unsigned short  ushort;/* System V compatibility */
typedef  unsigned int    uint;/* System V compatibility */

#ifdef vax
typedef  struct     _physadr { int r[1]; } *physadr;
typedef  struct     label_t{
         int        val[14];
} label_t;
#endif
#ifdef mc68000
typedef  struct     _physadr { short r[1]; } *physadr;
typedef  struct     label_t{
         int        val[13];
} label_t;
#endif
typedef  struct     _quad { long val[2]; } quad;
typedef  long       daddr_t;
typedef  char *     caddr_t;
typedef  u_long     ino_t;
typedef  long       swblk_t;
typedef  int        size_t;
typedef  long       time_t;
typedef  short      dev_t;
typedef  int        off_t;
typedef long        key_t;

typedef  struct     fd_set { int fds_bits[1]; } fd_set;
```

```
#endif
```

The form *daddr_t* is used for disk addresses, see *fs*(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(5), time(3C), lseek(2), adb(1)

NAME

utmp, wtmp, lastlog, usracct – login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file records information about who is currently using the system. The file is a sequence of entries with the following structure declared in the include file:

```
/*      @(#)utmp.h 1.1 86/07/07 SMI; from UCB 4.2 83/05/22      */

/*
 * Structure of utmp and wtmp files.
 *
 * Assuming the number 8 is unwise.
 */
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        char    ut_host[16];            /* host name, if remote */
        long    ut_time;        /* time on */
};

/*
 * This is a utmp entry that does not correspond to a genuine user
 */
#define nonuser(ut) ((ut).ut_host[0] == 0 &&        strncmp((ut).ut_line, "tty", 3) == 0 && ((ut).ut_line[3] ==
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(3C).

The *wtmp* file records all logins and logouts. A null user name indicates a logout on the associated terminal. Furthermore, the terminal name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed, record-keeping is turned off. It is summarized by *ac*(8).

/usr/adm/wtmp is appened to whenever a user logs in or out, and should be truncated periodically.

The *lastlog* file records the most recent login-date for every user logged in. When reporting (and updating) the most recent login date, *login*(1) performs an to a byte-offset in */usr/adm/lastlog* corresponding to the userid. Because the count of userids may be high, whereas the number actual users may be small within a network environment, the bulk of this file may never be allocated by the file system even though an offset may appear to be quite large. Although *ls* may show it to be large, chances are that this file need not truncated. *du*(1V) will report the correct (smaller) amount of space actually allocated to it.

The *usracct* file keeps login records in an older format.

FILES

/etc/utmp
/usr/adm/wtmp
/usr/adm/lastlog
/usr/adm/usracct

SEE ALSO

login(1), init(8), who(1), ac(8)

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1C)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word *begin* is followed by a mode (in octal), and a string which names the remote file. Spaces separate the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO

uuencode(1C), uudecode(1C), uusend(1C), uucp(1C), mail(1)

NAME

vfont – font formats

SYNOPSIS

#include <vfont.h>

DESCRIPTION

The fonts used by the window system and printer/plotters have the following format. Each font is in a file, which contains a header, an array of character description structures, and an array of bytes containing the bit maps for the characters. The header has the following format:

```
struct header {
        short           magic;          /* Magic number VFONT_MAGIC */
        unsigned short  size;           /* Total # bytes of bitmaps */
        short           maxx;           /* Maximum horizontal glyph size */
        short           maxy;           /* Maximum vertical glyph size */
        short           xtend;          /* (unused) */
};
#define VFONT_MAGIC                     0436
```

Maxx and *maxy* are intended to be the maximum horizontal and vertical size of any glyph in the font, in raster lines. (A glyph is just a printed representation of a character, in a particular size and font.) The *size* is the total size of the bit maps for the characters in bytes. The *xtend* field is not currently used.

After the header is an array of NUM_DISPATCH structures, one for each of the possible characters in the font. Each element of the array has the form:

```
struct dispatch {
        unsigned short  addr;           /* &(glyph) - &(start of bitmaps) */
        short           nbytes;         /* # bytes of glyphs (0 if no glyph) */
        char            up, down, left, right;  /* Widths from baseline point */
        short           width;          /* Logical width, used by troff */
};
#define NUM_DISPATCH                    256
```

The *nbytes* field is nonzero for characters which actually exist. For such characters, the *addr* field is an offset into the bit maps to where the character's bit map begins. The *up*, *down*, *left*, and *right* fields are offsets from the base point of the glyph to the edges of the rectangle which the bit map represents. (The imaginary "base point" is a point which is vertically on the "base line" of the glyph (the bottom line of a glyph which doesn't have a descender) and horizontally near the left edge of the glyph; often 3 or so pixels past the left edge.) The bit map contains *up+down* rows of data for the character, each of which has *left+right* columns (bits). Each row is rounded up to a number of bytes. The *width* field represents the logical width of the glyph in bits, and shows the horizontal displacement to the base point of the next glyph.

FILES

/usr/lib/vfont/*
/usr/lib/fonts/fixedwidthfonts/*

SEE ALSO

troff(1), pti(1), vfontinfo(1), vswap(1)

BUGS

A machine-independent font format should be defined. The **shorts** in the above structures contain different bit patterns depending whether the font file is for use on a Vax or a Sun. The *vswap* program must be used to convert one to the other.

NAME

vgrindefs – vgrind's language definition data base

SYNOPSIS

/usr/lib/vgrindefs

DESCRIPTION

Vgrindefs contains all language definitions for vgrind. The data base is very similar to *termcap*(5). Capabilities in *vgrindefs* are of two types: Boolean capabilities which indicate that the language has some particular feature and string capabilities which give a regular expression or keyword list. Entries may continue onto multiple lines by giving a \ as the last character of a line. Lines starting with # are comments.

Capabilities

The following table names and describes each capability.

Name	Type	Description
ab	str	Regular expression for the start of an alternate form comment
ae	str	Regular expression for the end of an alternate form comment
bb	str	Regular expression for the start of a block
be	str	Regular expression for the end of a lexical block
cb	str	Regular expression for the start of a comment
ce	str	Regular expression for the end of a comment
id	str	String giving characters other than letters and digits that may legally occur in identifiers (default '_')
kw	str	A list of keywords separated by spaces
lb	str	Regular expression for the start of a character constant
le	str	Regular expression for the end of a character constant
oc	bool	Present means upper and lower case are equivalent
pb	str	Regular expression for start of a procedure
pl	bool	Procedure definitions are constrained to the lexical level matched by the 'px' capability
px	str	A match for this regular expression indicates that procedure definitions may occur at the next lexical level. Useful for lisp-like languages in which procedure definitions occur as subexpressions of defuns.
sb	str	Regular expression for the start of a string
se	str	Regular expression for the end of a string
tc	str	Use the named entry as a continuation of this one
tl	bool	Present means procedures are only defined at the top lexical level

Regular Expressions

Vgrindefs uses regular expressions similar to those of *ex*(1) and *lex*(1). The characters '^', '$', ':', and '\' are reserved characters and must be 'quoted' with a preceding \ if they are to be included as normal characters. The metasymbols and their meanings are:

$	The end of a line
^	The beginning of a line
\d	A delimiter (space, tab, newline, start of line)
\a	Matches any string of symbols (like '.*' in lex)
\p	Matches any identifier. In a procedure definition (the 'pb' capability) the string that matches this symbol is used as the procedure name.
()	Grouping
|	Alternation
?	Last item is optional
\e	Preceding any string means that the string will not match an input string if the input string is preceded by an escape character (\). This is typically used for languages (like C) that can include the string delimiter in a string by escaping it.

Unlike other regular expressions in the system, these match words and not characters. Hence something like '(tramp|steamer)flies?' would match 'tramp', 'steamer', 'trampflies', or 'steamerflies'. Contrary to some forms of regular expressions, *vgrindef* alternation binds very tightly. Grouping parentheses are likely to be necessary in expressions involving alternation.

Keyword List

The keyword list is just a list of keywords in the language separated by spaces. If the 'oc' boolean is specified, indicating that upper and lower case are equivalent, then all the keywords should be specified in lower case.

EXAMPLE

The following entry, which describes the C language, is typical of a language entry.

```
C|c|the C programming language:\
        :pb=^\d?*?\d?\p\d??):bb={:be=}:cb=/*:ce=*/:sb=":se=\e":\
        :lb=':le=\e':tl:\
        :kw=asm auto break case char continue default do double else enum\
        extern float for fortran goto if int long register return short\
        sizeof static struct switch typedef union unsigned while #define\
        #else #endif #if #ifdef #ifndef #include #undef # define else endif\
        if ifdef ifndef include undef:
```

Note that the first field is just the language name (and any variants of it). Thus the C language could be specified to *vgrind*(1) as 'c' or 'C'.

FILES

/usr/lib/vgrindefs file containing terminal descriptions

SEE ALSO

vgrind(1), troff(1)

NAME

ypfiles – the yellowpages database and directory structure

DESCRIPTION

The yellow pages (YP) network lookup service uses a database of *dbm* files in the directory hierarchy at */etc/yp* . A *dbm* database consists of two files, created by calls to the *dbm*(3X) library package. One has the filename extension *.pag* and the other has the filename extension *.dir*. For instance, the database named *hosts.byname*, is implemented by the pair of files *hosts.byname.pag* and *hosts.byname.dir*. A *dbm* database served by the YP is called a YP *map*. A YP *domain* is a named set of YP maps. Each YP domain is implemented as a subdirectory of */etc/yp* containing the map. Any number of YP domains can exist. Each may contain any number of maps.

No maps are required by the YP lookup service itself, although they may be required for the normal operation of other parts of the system. There is no list of maps which YP serves - if the map exists in a given domain, and a client asks about it, the YP will serve it. For a map to be accessible consistently, it must exist on all YP servers that serve the domain. To provide data consistency between the replicated maps, an entry to run *ypxfr* periodically should be made in */usr/lib/crontab* on each server. More information on this topic is in ypxfr(8).

YP maps should contain two distinguished key-value pairs. The first is the key YP_LAST_MODIFIED, having as a value a ten-character ASCII order number. The order number should be the UNIX time in seconds when the map was built. The second key is YP_MASTER_NAME, with the name of the YP master server as a value. *makedbm* generates both key-value pairs automatically. A map that does not contain both key-value pairs can be served by the YP, but the *ypserv* process will not be able to return values for "Get order number" or "Get master name" requests. In addition, values of these two keys are used by *ypxfr* when it transfers a map from a master YP server to a slave. If *ypxfr* cannot figure out where to get the map, or if it is unable to determine whether the local copy is more recent than the copy at the master, you must set extra command line switches when you run it.

YP maps must be generated and modified only at the master server. They are copied to the slaves using *ypxfr*(8) to avoid potential byte-ordering problems among YP servers running on machines with different architectures, and to minimize the amount of disk space required for the dbm files. The YP database can be initially set up for both masters and slaves by using *ypinit*(8).

After the server databases are set up, it is probable that the contents of some maps will change. In general, some ASCII source version of the database exists on the master, and it is changed with a standard text editor. The update is incorporated into the YP map and is propagated from the master to the slaves by running */etc/yp/Makefile* . All Sun-supplied maps have entries in */etc/yp/Makefile* ; if you add a YP map, edit the this file to support the new map. The makefile uses *makedbm* to generate the YP map on the master, and *yppush* to propagate the changed map to the slaves. *yppush* is a client of the map *ypservers* , which lists all the YP servers. For more information on this topic, see yppush(8).

SEE ALSO

makedbm(8), ypinit(8), ypmake(8), ypxfr(8), yppush(8), yppoll(8), ypserv(8), rpcinfo(8),

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

U

V

W

X

Y

Z

Revision History

Version	Date	Comments
A	23 February 1983	First edition of this manual under the title *System Interface Manual for the Sun Workstation.*
B	15 April 1983	Second edition of this manual with corrections to numerous manual pages.
C	1 August 1983	Third edition of this manual with corrections to numerous manual pages. Added a glossary of system calls and system error responses.
D	1 November 1983	Fourth edition of this manual with numerous corrections. Corrected numerous incorrect cross-references. Added a *System Interface Overview* and the *Interprocess Communication Primer*.
E	7 January 1984	Fifth edition of this manual with numerous corrections.
F	15 May 1985	Sixth edition with numerous corrections. The *Interprocess Communication Primer* made a part of the manual, *Networking on the Sun Workstation*. Made page numbering contiguous throughout, and replaced the *Permuted Index* with a conventional one.

— *Continued*

Version	Date	Comments
G	1 January 1986	Formerly the *System Interface Manual for the Sun Workstation*, this seventh edition contains many corrections to manual pages. The former section entitled *System Interface Overview* is now a separate manual entitled *UNIX Interface Overview*. The index has been upgraded to refer to `ioctl`'s and system error numbers.
H	15 October 1986	Eighth (draft) edition, for Sun 3.2 Release. Includes numerous additions for System V compatibility, and updates from U.C. Berkeley 4.3 BSD, as well as numerous corrections to manual pages.

Notes

Notes