PLAN 9
from
BELL LABS

PROGRAMMER'S MANUAL

First Edition

Computing Science Research Center
AT&T Bell Laboratories
Murray Hill, New Jersey

PERMUTED INDEX

Manual pages for all sections are accessible on line through *man*(1).

To save space, neighboring references to the same page have been collapsed into a single reference. This should cause no difficulty in cases like ‘atan’ and ‘atan2’, but is somewhat obscure in the case of ‘strcat’ and ‘strchr’.

NAME

intro − introduction to Plan 9

DESCRIPTION

Plan 9 is a distributed computing environment assembled from separate machines acting as terminals, CPU servers, and file servers. A user works at a terminal, running a window system on a bitmapped display. Some windows are connected to CPU servers; the intent is that heavy computing should be done in those windows but it is also possible to compute on the terminal. A separate file server provides file storage for terminals and CPU servers alike.

Name Spaces

In Plan 9, almost all objects look like files. The object retrieved by a given name is determined by a mapping called the *name space* . A quick tour of the standard name space is in *namespace*(4). Every program running in Plan 9 belongs to a *process group* (see *rfork* in *fork*(2)), and the name space for each process group can be independently customized.

A name space is hierarchically structured. A full file name (also called a *full path name*) has the form

> */e1/e2/.../en*

This represents an object in a tree of files: the tree has a root, represented by the first `/`; the root has a child file named *e1*, which in turn has child *e2*, and so on; the descendent *en* is the object represented by the path name.

There are a number of Plan 9 *services* available, each of which provides a tree of files. A name space is built by *binding* services (or subtrees of services) to names in the name-space-so-far. Typically, a user's home file server is bound to the root of the name space, and other services are bound to conventionally named subdirectories. For example, there is a service resident in the operating system for accessing hardware devices and that is bound to `/dev` by convention. Kernel services have names (outside the name space) that are a `#` sign followed by a single letter; for example, `#c` is conventionally bound to `/dev`.

Plan 9 has *union directories*: directories made of several directories all bound to the same name. The directories making up a union directory are ordered in a list. When the bindings are made (see *bind*(1)), flags specify whether a newly bound member goes at the head or the tail of the list or completely replaces the list. To look up a name in a union directory, each member directory is searched in list order until the name is found. A bind flag specifies whether file creation is allowed in a member directory: a file created in the union directory goes in the first member directory in list order that allows creation, if any.

The glue that holds Plan 9 together is a network protocol called *9P*, described in section 5 of this manual. All Plan 9 servers read and respond to 9P requests to navigate through a file tree and to perform operations such as reading and writing files within the tree.

Booting

When a terminal is powered on or reset, it must be told the name of a file server to boot from, the operating system kernel to boot, and a user name and password. How this dialog proceeds is environment- and machine-dependent. Once it is complete, the terminal loads a Plan 9 kernel, which sets some environment variables (see *env*(3)) and builds an initial name space. See *namespace*(4), *boot*(8), and *init*(8) for details, but some important aspects of the initial name space are:

- The environment variable `$cputype` is set to the name of the kernel's CPU's architecture: one of `68020`, `mips`, `sparc`, `386`, or `hobbit`. The environment variable `$objtype` is initially the same as `$cputype`.
- The environment variable `$terminal` is set to the model of the machine running the kernel: e.g., `mips magnum 3000`.
- The environment variable `$service` is set to `terminal`. (Other ways of accessing Plan 9 may set `$service` to one of `cpu`, `con`, or `rx`.)
- The environment variable `$user` is set to the name of the user who booted the terminal. The environment variable `$home` is set to that user's home directory.

- `/$cputype/bin` and `/rc/bin` are unioned into `/bin`.

After booting, the terminal runs the command interpreter, *rc*(1), on `/usr/$user/lib/profile` after moving to the user's home directory.

Here is a typical profile:

```
bind -c $home/tmp /tmp
bind -a $home/bin/rc /bin
bind -a $home/bin/$cputype /bin
font = /lib/font/bit/pelm/latin1.9.font
switch($service){
case terminal
      prompt=('term% ' '      ')
      exec 8½ -f $font
case cpu
      bind -b /mnt/term/mnt/8½ /dev
      prompt=('cpu% ' ' ')
      news
case con
      prompt=('cpu% ' ' ')
      news
}
```

The first three lines replace `/tmp` with a `tmp` in the user's home directory and union personal `bin` directories with `/bin`, to be searched after the standard `bin` directories. Then different things happen, depending on the `$service` environment variable, such as running the window system *8½*(1) on a terminal.

To do heavy work such as compiling, the *cpu*(1) command connects a window to a CPU server; the same environment variables are set (to different values) and the same profile is run. The initial directory is the current directory in the terminal window where *cpu* was typed. The value of `$service` will be `cpu`, so the second arm of the profile switch is executed. The root of the terminal's name space is accessible through `/mnt/term`, so the *bind* is a way of making the window system's graphics interface (see *bit*(3)) available to programs running on the CPU server. The *news*(1) command reports current Plan 9 affairs.

The third possible service type, `con`, is set when the CPU server is called from a non-Plan-9 machine, such as through *telnet* (see *con*(1)).

Using Plan 9

The user commands of Plan 9 are reminiscent of those in Research Unix, version 10; the window system is a lot like *mux*. There are a number of differences, however.

The standard shell is *rc*(1), not the Bourne shell. The most noticeable differences appear only when programming and macro processing.

The character-delete character is backspace, and the line-kill character is control-U; these cannot be changed.

DEL is the interrupt character. The shell kills any running commands if you type a DEL in its window. See *keyboard*(6) for instructions on typing characters like DEL on the various keyboards.

If a program dies with something like an address error, it enters a 'Broken' state. It lingers, available for debugging with *db*(1). *Broke* (see *kill*(1)) cleans up broken processes.

The standard editor is *sam*(1). There is a variant that permits running the file-manipulating part of *sam* on a non-Plan-9 system:

```
sam -r tcp!kremvax
```

Machine names may be prefixed by the network name, here `tcp`; others include `dk` for Datakit and `il` for the Plan 9 Internet protocol.

Login connections and remote execution on non-Plan-9 machines are usually done by saying, for example,

```
con kremvax
```

or

```
rx deepthought chess
```

(see *con*(1)).

You can access file systems of other machines by using *9fs* (see *srv*(4)). For example,

```
9fs kremvax
```

sets things up so that the root of `kremvax`'s file tree is visible locally in `/n/kremvax`.

You can get notification of mail arriving on Plan 9 using *seemail* (see *mail*(1)); if your mail arrives elsewhere, use *vismon*:

```
vismon tcp!kremvax
```

The Plan 9 file server has an integrated backup facility. The command

```
9fs dump
```

binds to `/n/dump` a tree containing the daily backups on the file server. The dump tree has years as top level file names, and month-day as next level file names. For example, `/n/dump/1990/0120` is the root of the file system as it appeared at dump time on January 20, 1990. If more than one dump is taken on the same day, dumps after the first have an extra digit. To recover the version of this file as it was on June 15, 1991,

```
cp /n/dump/1991/0615/sys/man/man1/Intro.1 .
```

or use *yesterday*(1).

SEE ALSO

This section for general publicly accessible commands.
Section (2) for library functions, including system calls.
Section (3) for kernel devices (accessed via *bind*(1)).
Section (4) for file services (accessed via *mount*).
Section (5) for the Plan 9 file protocol.
Section (6) for file formats.
Section (7) for databases and database access programs.
Section (8) for things related to administering Plan 9.
Section (9) for raster image software.
Section (10) for circuit design software.
`/sys/doc` for copies of papers referenced in this manual.

DIAGNOSTICS

Upon termination each program returns a string called the *exit status*. It was either supplied by a call to *exits*(2) or was written to the command's `/proc/`*pid*`/note` file (see *proc*(3)), causing an abnormal termination. The empty string is customary for successful execution; a non-empty string gives a clue to the failure of the command.

NAME

2a, 6a, 8a, ka, va, za − assemblers

SYNOPSIS

`2a` [*option ...*] [*name ...*]
`6a` [*option ...*] [*name ...*]
`8a` [*option ...*] [*name ...*]
`ka` [*option ...*] [*name ...*]
`va` [*option ...*] [*name ...*]
`za` [*option ...*] [*name ...*]

DESCRIPTION

2a, *6a*, *8a*, *ka*, *va*, and *za* assemble the named files into MC68020, i960, i386, SPARC, MIPS, and Hobbit object files. The assemblers handle the most common C preprocessor directives and associated command-line options. Other options are:

`-o` *obj* Place output in file *obj* (allowed only if there is just one input file). Default is to take the last element of the input path name, strip any trailing `.s`, and append *.O*, where *O* is first letter of the assembler's name.

FILES

The directory `/sys/include` is searched for include files after machine-dependent files in `/$objtype/include`.

SEE ALSO

2c(1), *2l*(1).

Rob Pike, ‘‘A manual for the Plan 9 assembler.’’

NAME

2c, 6c, 8c, kc, vc, zc − C compilers

SYNOPSIS

`2c` [*option* ...] [*name* ...]
`6c` [*option* ...] [*name* ...]
`8c` [*option* ...] [*name* ...]
`kc` [*option* ...] [*name* ...]
`vc` [*option* ...] [*name* ...]
`zc` [*option* ...] [*name* ...]

DESCRIPTION

2c, *6c*, *8c*, *kc*, *vc*, and *zc* compile the named C files into MC68020, i960, i386, SPARC, MIPS, and Hobbit object files. The compilers handle most preprocessing directives themselves; a complete preprocessor is available in *cpp*(1), which must be run separately.

Let the first letter of the compiler name be *O*= `2`, `6`, `8`, `k`, `v`, or `z`. The output object files end in `.`*O*. The letter is also the prefix of related programs: *O*`a` is the assembler, *O*`l` is the loader. Associated with each compiler is a string *objtype*= `68020`, `960`, `386`, `sparc`, `mips`, or `hobbit`. Plan 9 conventionally sets the `$objtype` environment variable to the *objtype* string appropriate to the current machine's type. Plan 9 also conventionally has /*objtype* directories, which contain among other things: `include`, for machine-dependent include files; `lib`, for public object code libraries; `bin`, for public programs; and `mkfile`, for preconditioning *mk*(1).

The compiler options are:

`-o` *obj*	Place output in file *obj* (allowed only if there is just one input file). Default is to take the last element of the input pathname, strip any trailing `.c`, and append `.`*O*.
`-w`	Print warning messages about unused variables, etc.
`-A`	Complain about functions used without a new-style ANSI C function prototype. This option is on by default.
`-B`	Turn off the action of the `-A` flag.
`-D`*name=def* `-D`*name*	Define the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as `1`.
`-I`*dir*	`#include` files whose names do not begin with `/` are always sought first in the directory of the *file* argument, then in directories named in `-I` options, then in `/sys/include`, and finally in `/$objtype/include`.
`-O`	Perform object code optimization. This option is on by default.
`-N`	Turn off the action of the `-O` flag.
`-S`	Print an assembly language version of the object code on standard output as well as generating the `.`*O* file.
`-s`*name*	Print on standard output a listing of the fields in structure or union *name* together with their offsets and some type information. This can be used in conjunction with the debugger (see *db*(1)).

The compilers support several extensions to ANSI C:

- A structure or union may contain unnamed substructures and subunions. The fields of the substructures or subunions can then be used as if they were members of the parent structure or union (the resolution of a name conflict is unspecified). When a pointer to the outer structure or union is used in a context that is only legal for the unnamed substructure, the compiler promotes the type and adjusts the pointer value to point at the substructure.

- A structure value can be formed with an expression such as
  `(struct S){v1, v2, v3}`
  where the list elements are values for the fields of struct `S`.
- Array initializers can specify the indices of the array in square brackets, as
  `int a[] = { [3] 1, [10] 5 };`
  which initializes the third and tenth elements of the eleven-element array `a`.
- A global variable can be dedicated to a register by declaring it `extern register` in *all* modules and libraries.
- A `#pragma` of the form
  `#pragma lib "libbio.a"`
  records that the program needs to be loaded with file `/$objtype/lib/libbio.a`; such lines, typically placed in library header files, obviate the `-l` option of the loaders.

EXAMPLE

For the 68020, produce a program `prog` from C files `main.c` and `sub.c`:

```
2c -w main.c sub.c
2l -o prog main.2 sub.2
```

FILES

`/sys/include`	system area for machine-independent `#include` directives.
`/$objtype/include`	system area for machine-dependent `#include` directives.

SEE ALSO

2a(1), *2l*(1), *rl*(1), *mk*(1), *nm*(1), *db*(1)

Rob Pike, ‘‘How to Use the Plan 9 C Compiler’’

BUGS

The i960 compiler has been used only to program one I/O controller and is certainly buggy.

Bitfields are not supported in *zc*.

Unsigned integers as large as 2^31 are not correctly converted to floating.

The preprocessor only handles `#define`, `#include`, `#undef`, `#ifdef`, `#line`, and `#ifndef`. For a full ANSI preprocessor, use *cpp*(1) on the files first.

NAME

2l, 6l, 8l, kl, vl, zl − loaders

SYNOPSIS

`2l` [*option* ...] [*name* ...]
`6l` [*option* ...] [*name* ...]
`8l` [*option* ...] [*name* ...]
`kl` [*option* ...] [*name* ...]
`vl` [*option* ...] [*name* ...]
`zl` [*option* ...] [*name* ...]

DESCRIPTION

2l, *6l*, *8l*, *kl*, *vl*, and *zl* load the named files into MC68020, i960, i386, SPARC, MIPS, and Hobbit executable files. The files should be object files or libraries (archives of object files) for the appropriate architecture. Also, a name like `-l`*ext* represents the library `lib`*ext*`.a` in `/$objtype/lib`, where *objtype* is one of `68020`, `960`, `386`, `sparc`, `mips`, or `hobbit` . In practice, such options are rarely necessary as the header files for the libraries cause their archives to be included automatically in the load (see *2c*(1)). The libraries must have tables of contents (see *rl*(1)).

Normally there is an implicit `-lc` after the named files to search the C library `/$objtype/lib/libc.a`. Also, the loader creates an undefined symbol `_main` (or `_mainp` if profiling is enabled) to force loading of the startup linkage from the C library.

The loader options are:

`-l`	(As a bare option.) Suppress the default loading of the C library and startup linkage.
`-o` *out*	Place output in file *out.* Default is *O*`.out,` where *O* is the first letter of the loader name.
`-p`	Insert profiling code into the executable output.
`-s`	Strip the symbol tables from the output file.
`-a`	Print the object code in assembly language, with addresses.
`-v`	Print debugging output that annotates the activities of the load.
`-H`*n*	Executable header is type *n*. The meaning of the types is architecture-dependent; typically type 1 is Plan 9 boot format and type 2 is the regular Plan 9 format, the default. These are reversed on the MIPS. The Next boot format is 3.
`-T`*t*	The text segment starts at address *t.*
`-D`*d*	The data segment starts at address *d*.
`-R`*r*	The text segment is rounded to a multiple of *r* (if *r* is nonzero).

The numbers in the above options can begin with `0x` or `0` to change the default base from decimal to hexadecimal or octal. The defaults for the values depend on the compiler and the header type.

FILES

`/$objtype/lib` for `-l`*lib* arguments.

SEE ALSO

2c(1), *2a*(1), *ar*(1), *rl*(1), *nm*(1), *db*(1), *prof*(1)

Rob Pike, ‘‘How to Use the Plan 9 C Compiler’’

NAME

8½, label, window, wloc − window system

SYNOPSIS

`8½` [`-i` *'cmd'*] [`-s`] [*font*]

`label` *name*

`window` *' minx miny maxx maxy'* *cmd arg* . . .

`wloc`

DESCRIPTION

8½ manages asynchronous layers of text, or windows, on a bit-mapped display. It also serves a variety of files for communicating with and controlling windows; these are discussed in section *8½*(4).

Commands

The *8½* command starts a new instance of the window system. Its `-i` option names a startup script, which typically contains several *window* commands generated by *wloc*.

The `-s` option initializes windows so that text scrolls; the default is not to scroll. The *font* argument names a font used to display text, both in *8½*'s menus and as a default for any programs running in its windows; it also establishes the environment variable `$font`. If `-f` is not given, *8½* uses the imported value of `$font` if set; otherwise it imports the default font from the underlying graphics server, usually the terminal's operating system.

The *label* command changes a window's identifying name.

The *window* command creates a window. The first argument gives the minimum and maximum screen coordinates of the window to be created; the rest of the arguments are the command to be run in the window and its arguments.

The *wloc* command prints the coordinates and label of each window in its instance of *8½* and is used to construct arguments for *window*.

Window control

Each window behaves as a separate terminal with at least one process associated with it. When a window is created, a new process (usually a shell; see *rc*(1)) is established and bound to the window as a new process group. Initially, each window acts as a simple terminal that displays character text; the standard input and output of its processes are attached to `/dev/cons`. Other special files, accessible to the processes running in a window, may be used to make the window a more general display. Some of these are mentioned here; the complete set is discussed in *8½*(4).

One window is *current*, and is highlighted with a heavy border; characters typed on the keyboard are available in the `/dev/cons` file of the process in the current window. Characters written on `/dev/cons` appear asynchronously in the associated window whether or not the window is current.

Windows are created, deleted and rearranged using the mouse. Clicking (depressing and releasing) mouse button 1 in a non-current window makes that window current and brings it in front of any windows that happen to be overlapping it. When the mouse cursor points to the background area or is in a window that has not claimed the mouse for its own use, depressing mouse button 3 activates a menu of window operations provided by *8½*. Releasing button 3 then selects an operation. At this point, a gunsight or cross cursor indicates that an operation is pending. The button 3 menu operations are:

`New`	Create a window. Depress button 3 where one corner of the new rectangle should appear (cross cursor), and move the mouse, while holding down button 3, to the diagonally opposite corner. Releasing button 3 creates the window, and makes it current. Very small windows may not be created.
`Reshape`	Change the size and location of a window. First click button 3 in the window to be changed (gunsight cursor). Then sweep out a window as for the `New` operation. The window is made current.

`Move`	Move a window to another location. Depress button 3 in one quadrant of the window to be moved (cross cursor), then move the mouse, while holding down button 3, to the place where the indicated quadrant's corner should appear. The window is made current.
`Delete`	Delete a window. Click in the window to be deleted (gunsight cursor). Deleting a window causes a `hangup` note to be sent to all processes in the window's process group (see *notify*(2)).
`Hide`	Hide a window. Click in the window to be hidden (gunsight cursor); it will be moved off-screen. Each hidden window is given a menu entry in the button 3 menu according to the value of the file `/dev/label`, which *8½* maintains (see *8½*(4)).

Text windows

Characters typed on the keyboard or written to `/dev/cons` collect in the window to form a long, continuous document.

There is always some *selected text,* a contiguous string marked on the screen by reversing its color. If the selected text is a null string, it is indicated by a hairline cursor between two characters. The selected text may be edited by mousing and typing. Text is selected by pointing and clicking button 1 to make a null-string selection, or by pointing, then sweeping with button 1 depressed. Text may also be selected by double-clicking: just inside a matched delimiter-pair with one of `` { [ (<«`'" `` on the left and `` } ] ) >»`'" `` on the right, it selects all text within the pair; at the beginning or end of a line, it selects the line; within or at the edge of an alphanumeric word, it selects the word.

Characters typed on the keyboard replace the selected text; if this text is not empty, it is placed in a *snarf buffer* common to all windows but distinct from that of *sam*(1).

Programs access the text in the window at a single point maintained automatically by *8½.* The *output point* is the location in the text where the next character written by a program to `/dev/cons` will appear; afterwards, the output point is the null string beyond the new character. The output point is also the location in the text of the next character that will be read (directly from the text in the window, not from an intervening buffer) by a program from `/dev/cons`. When such a read will occur is, however, under control of *8½* and the user.

In general there is text in the window after the output point, usually placed there by typing but occasionally by the editing operations described below. A pending read of `/dev/cons` will block until the text after the output point contains a newline, whereupon the read may acquire the text, up to and including the newline. After the read, as described above, the output point will be at the beginning of the next line of text. In normal circumstances, therefore, typed text is delivered to programs a line at a time. Changes made by typing or editing before the text is read will be seen by the program reading it. If the program in the window does not read the terminal, for example if it is a long-running computation, there may accumulate multiple lines of text after the output point; changes made to all this text will be seen when the text is eventually read. This means, for example, that one may edit out newlines in unread text to forestall the associated text being read when the program finishes computing. This behavior is very different from most systems'.

Even when there are newlines in the output text, *8½* will not honor reads if the window is in *hold mode,* which is indicated by a white cursor and border. The ESC character toggles hold mode. Some programs, such as *mail*(1), automatically turn on hold mode to simplify the editing of multi-line text; type ESC when done to allow *mail* to read the text.

An EOT character (control-D) behaves exactly like newline except that it is not delivered to a program when read. Thus on an empty line an EOT serves to deliver an end-of-file indication: the read will return zero characters. Like newlines, unread EOTs may be successfully edited out of the text. The BS character (control-H) erases the character before the selected text. The ETB character (control-W) erases any nonalphanumeric characters, then the alphanumeric word just before the selected text. 'Alphanumeric' here means non-blanks and non-punctuation. The NAK character (control-U) erases the text after the output point, and not yet read by a program, but not more than one line. All these characters are typed on the keyboard and hence replace the selected text; for example, typing a BS with a word selected places the word in the snarf buffer, removes it from the screen, and erases the character before the word.

Text may be moved vertically within the window. A scroll bar on the left of the window shows in its clear portion what fragment of the total output text is visible on the screen, and in its gray part what is above or below view; it measures characters, not lines. Mousing inside the scroll bar moves text: clicking button 1 with the mouse pointing inside the scroll bar brings the line at the top of the window to the cursor's vertical location; button 3 takes the line at the cursor to the top of the window; button 2, treating the scroll bar as a ruler, jumps to the indicated portion of the stored text. Also, a VIEW key (possibly with a different label; see *keyboard*(6)) scrolls forward half a window.

The DEL character sends an `interrupt` note to all processes in the window's process group. Alone among characters, the DEL and VIEW keys do not snarf the selected text.

Normally written output to a window blocks when the text reaches the end of the screen; a button 2 menu item toggles scrolling.

Other editing operations are selected from a menu on button 2. The `cut` operation deletes the selected text from the screen and puts it in the snarf buffer; `snarf` copies the selected text to the buffer without deleting it; `paste` replaces the selected text with the contents of the buffer; and `send` copies the snarf buffer to just after the output point, adding a final newline if missing. `Paste` will sometimes and `send` will always place text after the output point; the text so placed will behave exactly as described above. Therefore when pasting text containing newlines after the output point, it may be prudent to turn on hold mode first.

Raw text windows

Opening or manipulating certain files served by *8½* suppresses some of the services supplied to ordinary text windows. While the file `/dev/mouse` is open, any mouse operations are the responsibility of another program running in the window. Thus, *8½* refrains from maintaining the scroll bar, supplying text editing or menus, interpreting the VIEW key as a request to scroll, and also turns scrolling on.

The file `/dev/consctl` controls interpretation of keyboard input. In particular, a raw mode may be set: in a raw-input window, no typed keyboard characters are special, they are not echoed to the screen, and all are passed to a program immediately upon reading, instead of being gathered into lines.

Graphics windows

A program that holds `/dev/mouse` and `/dev/consctl` open after putting the console in raw mode has complete control of the window: it interprets all mouse events, gets all keyboard characters, and determines what appears on the screen.

FILES

`/lib/font/bit/*`	font directories
`/mnt/8½`	Files served by *8½* (also unioned in `/dev` in a window's name space, before the terminal's real `/dev` files)
`/srv/8½.`*user*`.`*pid*	Server end of *8½*.

SEE ALSO

8½(4), *rc*(1), *cpu*(1), *sam*(1), *mail*(1), *proof*(1), *graphics*(2), *frame*(2), *layer*(2), *notify*(2), *cons*(3), *bit*(3), *keyboard*(6)

NAME

acid − debugger

SYNOPSIS

`acid` [*-l loadmodule*] [*-w*] [*pid*] [*textfile*]

DESCRIPTION

Acid is a general purpose source level, symbolic debugger. The debugger is built around a simple command language. The command language provides a flexible user interface which allows the debugger interface to be customized for a specific application or architecture. Moreover, it provides an opportunity to write test and verification code independently of a programs source code. Acid is able to debug multiple processes provided they share a common set of symbols (See ALEF(1)).

The -w option allows the textfile to be modified.

Control-D terminates the program. If there are active processes a diagnostic is printed and acid returns to the prompt. A second Control-D will cause the debugger to exit and attached processes will be unaffected. To make a clean exit, processes should be destroyed using the function *kill* .

At startup acid reads a standard set of command functions from a library. Modules are automatically loaded from /lib/acid/port, /lib/acid/$objtype and $home/lib/acid in that order. This provides a standard debugging environment for each of the architectures. Definitions in $home/lib/acid may replace any previously defined functions. Language specific modules can also be loaded using a command line option. Modules specified on the command line are loaded last. If the function acidinit is defined by any of the load modules it will be invoked after all modules have been loaded.

Acid introduces the symbols of the program being debugged as variables in the language. If a symbol in the program conflicts with a predefined variable or reserved word the symbol is renamed. The interpreter prepends $ characters to start of the symbol until it is unique. A summary of the renamings is printed at startup.

Acid has an expression syntax much like C. However since the symbol table provides addresses the dual of a program variable will be an address in acid. That is the same as if all names were preceded by an & in C or ALEF expression. To obtain the value of a variable one of the indirection operators must be used.

At the prompt acid is prepared to either store function definitions or evaluate expressions. The expression syntax is similar to the expression syntax of C and ALEF. The result of expression evaluation yields both a value and a format. The format of and item may be set using the builtin function fmt. Formats are compatible with db(1). The format determines how an item will be printed, the stride of an increment or decrement operation and how many bytes are read or written by the indirection operators.

EXAMPLE

To start to debug ls and set a breakpoint:

```
% acid /bin/ls
/bin/ls: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
acid: new()
70094 : system call     _main ADD   $-14,R29
acid: bpset(ls)
acid: cont()
70094 : breakpoint      ls    ADD   $-16c8,R29
acid:
```

FILES

```
/proc/*/text
/proc/*/mem
/proc/*/ctl
/proc/*/note
```

```
/lib/acid/$objtype
/lib/acid/port
$home/lib/acid
```

SEE ALSO

2a(1), *2l*(1), *mk*(1), *db*(1)
ACID Manual, Phil Winterbottom

NAME

val, kal − ALEF compilers

SYNOPSIS

`kal` [*option* ...] [*name* ...]
`val` [*option* ...] [*name* ...]

DESCRIPTION

ALEF is a concurrent programming language with a syntax like C's. *Kal* and *val* compile the named ALEF source files into SPARC and MIPS object files. Source files have the extension `.l`. The ALEF source is passed through *cpp*(1) prior to compilation. Object files have the normal extension for each architecture: `.k` for SPARC and `.v` for MIPS.

The compiler options are:

`−o` *obj*	Place output in file *obj* (ignored if there is more than one input file). Default is to take the last element of the input pathname, strip any trailing `.l`, and append `.v` or `.k` .
`−w`	Print warning messages for non fatal errors.
`−N`	Do not run the code optimizer.
`−c`	Generate code for `check` statements.
`−S`	Produce assembly language as output. Default is to take the last element of the input pathname, strip any trailing `.l`, and append `.s`.
`−Idir`	The directory *dir* is added to the front of the include search path.
`−D`*name=def* `−D`*name*	Define the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as `1`.
`−I`*dir*	`#include` files whose names do not begin with `/` are always sought first in the directory of the *file* argument, then in directories named in `−I` options, then in `/sys/include/alef`, and finally in `/$objtype/include/alef.`
`−d#`	Produce various forms of debugging. The # is a character in the range a-z or A-Z.

EXAMPLE

To compile and run on a SPARC the ALEF program in the current directory:

```
kal -w *.l
kl *.k
k.out
```

FILES

`/sys/include/alef`	directory for `#include` files.
`/$objtype/lib/alef`	directory for ALEF libraries

SEE ALSO

2a(1), *2l*(1), *rl*(1), *mk*(1), *nm*(1), *db*(1)
ALEF Reference Manual, Phil Winterbottom

NAME

ar − archive and library maintainer

SYNOPSIS

`ar` *key* [*posname*] *afile* [*file* ...]

DESCRIPTION

Ar maintains groups of files combined into a single archive file, *afile*. The main use of *ar* is to create and update library files for the loaders *2l*(1), etc. It can be used, though, for any similar purpose.

Key is one character from the set `drqtpmx`, optionally concatenated with one or more of `vuaibcl`. The *files* are constituents of the archive *afile.* The meanings of the *key* characters are:

`d` Delete *files* from the archive file.

`r` Replace *files* in the archive file. Optional modifiers are
- `u` Replace only files with modified dates later than that of the archive.
- `a` Place new files after *posname* in the archive rather than at the end.
- `b` or `i` Place new files before *posname* in the archive.

`q` Quick. Append *files* to the end of the archive without checking for duplicates. Avoids quadratic behavior in `for (i in *.o) ar r lib.a $i`.

`t` List a table of contents of the archive. If names are given, only those files are listed.

`p` Print the named files in the archive.

`m` Move the named files to the end or elsewhere, specified as with `r`.

`x` Extract the named files. If no names are given, all files in the archive are extracted. In neither case does `x` alter the archive file.

`v` Verbose. Give a file-by-file description of the making of a new archive file from the old archive and the constituent files. With `p`, precede each file with a name. With `t`, give a long listing of all information about the files, somewhat like a listing by *ls*(1), showing

```
mode uid/gid size date name
```

`c` Create. Normally *ar* will create a new archive when *afile* does not exist, and give a warning. Option `c` discards any old contents and suppresses the warning.

`l` Local. Normally *ar* places its temporary files in the directory `/tmp`. This option causes them to be placed in the local directory.

EXAMPLE

```
ar cr lib.a *.o; rl lib.a
```

Replace the contents of library `lib.a` with the object files in the current directory.

FILES

`/tmp/v*`	temporaries

SEE ALSO

2l(1), *ar*(6)

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

This command predates Plan 9 and makes some invalid assumptions, for instance that user id's are numeric.

NAME

art, art2pic − edit line-art

SYNOPSIS

`art` [*font*]

`art2pic` *file*

DESCRIPTION

Art is an interactive program to create precise line-art in the style of *pic*(1). It is mostly mouse-operated, with a few commands entered from the keyboard. It divides its layer into four areas: a menu bar at the top, a one-line rectangle immediately below for echoing typed-in characters, another one-line rectangle below that for printing messages, and a large area at the bottom for displaying the drawing.

A small caret whose apex is the *current point*, appears on the screen. The lines, arcs and other elements of a drawing are constructed by placing the caret at each of a sequence of points that define the item. When the caret moves, two other markers (a square and a cross) trail it, showing its previous positions. Items on the screen near the caret attract it. Important points on items, like endpoints or intersections, pull harder than more mundane points, making precise alignment easy.

On request, *art* will automatically construct alignment lines and circles, which it displays more faintly than items in the drawing. Certain lines and points in a drawing are *hot*. *Art* constructs circles of given radii and lines of given slopes at hot points, and parallels at given distances from hot lines and lines at given angles to their endpoints. Menus pulled down from the menu bar control what alignment items are constructed. Items are automatically heated when added to the drawing and will be heated or cooled on command.

The ‘important points’ on a line segment to which the caret preferentially gravitates are its endpoints and midpoint. On a circle, the center is important. On an arc, the endpoints are important. On a box, the corners, the midpoints of the sides and the center are important. Likewise, on a piece of text, the corners, midpoints and center of its bounding box are important. On a spline, the control points are important, and in a group, the important points of the group members are important.

Whenever button 1 is pressed, the caret follows the mouse cursor. On release, the item the caret is touching, if any, is selected and highlighted. If more than one item touches the caret, clicking button 1 repeatedly will cycle through them.

Pressing button 2 pops up a menu of commands that add new items to the drawing. Every item is described by several *control points*: a line by its endpoints, a circular arc by its endpoints and a third point on the arc, and so forth. A new item is specified by moving the caret in turn to each control point but the last, selecting a menu entry with button two, then using button 1 to place the caret on the last control point. (Buttons 2 and 3 will cancel the command.) While the caret is being dragged to the last control point, *art* displays and updates the item on the screen (‘‘rubber-banding’’). In all cases, after making an addition to the drawing, the new item becomes the current selection. The button 2 menu operations are

`line`	Add a line segment to the drawing. The two control points are the segment’s endpoints.
`circle`	Add a circle to the drawing. The first control point is the center. The second is a point on the circumference.
`arc`	Add a circular arc to the drawing. The endpoints are the first and third control points. The second control point is an interior point of the arc.
`box`	Add a rectangle to the drawing. The box’s sides are vertical and horizontal. The two control points are two diagonally opposite corners.
`spline`	Add a spline curve to the drawing, or extend an existing spline. Splines are a little more complicated than other items because they may have any number of control points. If the current selection is not a spline, there are two control points — the ends of a new spline. When a spline is selected, the `spline` button adds a new control point to the end of the spline closest to the selected point.
`group`	Add a group to the drawing. The two control points are the diagonally opposite corners of a rectangle. Any item partially or completely contained in the rectangle is made part of the

group. Henceforth the group acts as a monolithic item and may be moved, deleted or copied as a unit. The `open`, `close` and `flatten` commands (on button 3) allow manipulation of the items within a group. Groups may be nested within other groups.

Button 3's menu has commands to manipulate the current selection.

`delete`	Remove the selection from the drawing. If the whole drawing is selected, *art* asks for confirmation by pressing mouse button 3. Buttons 1 and 2 cancel the command.
`heat`	Heat the selected item.
`copy`	Create a duplicate of the selected item. The duplicate must be dragged to its intended position using button 1. Buttons 2 and 3 cancel the command.
`edit`	Change the indicated point of the selected item. Button 1 adjusts the point. Buttons 2 and 3 cancel the command. This command's behavior depends on the kind of item and the point at which it is selected. If a line is selected near an endpoint, that endpoint moves and the other remains fixed. Both endpoints of a line selected near its midpoint move—its length and slope do not change. If a circle selected at its center, it translates without changing its radius. If selected on its circumference, its radius changes but its center remains fixed. The control point of an arc or spline nearest the selection point is modified. If a box is selected near a corner, that corner moves and the other remains fixed. If selected near the middle of an edge, the edge moves, but the opposite edge remains fixed. If selected near its center, the whole box moves without changing its size or shape. A group or a piece of text translates, regardless of the selection point.
`open`	The selection must be a group. All commands now operate on the members of the group.
`close`	The most-recently opened group is closed. Any changes made while it was open are propagated to other copies.
`flatten`	The selection must be a group. The items in the group are inserted in its place in the drawing. This undoes the effect of the `group` command. Other copies of the group are unaffected.

Keyboard commands:

`t text`	Add text to the drawing. The text is in the current font and is drawn centered on the caret.
`f` *name*	Set the current typeface. *Name* is the name of a font file. Subdirectories of `/lib/font/bit` contain many appropriate fonts.
`D`	Redraw the display.
`q`	Quit. *Art* exits, without asking for confirmation.
`w` *[file]*	Write the drawing into a file in *art* format. *File* defaults to the last file mentioned in a read or write command. *Art* files may be converted to *pic*(1) format by the *art2pic* command.
`r` *[file]*	Read a drawing from a file. *File* defaults to the last file mentioned in a read or write command.
`c`	Cool everything. Every hot item is cooled.
`a`	Select all items. The entire drawing is selected.
`d`	Drop anchor. The *anchor* is the fixed point for the not-yet-implemented rotate and scale commands.

Menus pulled down from the menu bar contain commands that alter how *art* responds to user interaction.

`slopes`	Most of the entries in this menu are numbers, representing angles in degrees. Those that are marked with a star are the inclination from horizontal of alignment lines constructed at each hot point of the drawing. Selecting a number toggles the star on and off. The `measure` button measures the slope of the line connecting the most recent two points selected with the caret. The measurement is printed, and a corresponding new entry is made in the menu.
`angles`	This menu behaves much like the `slopes` menu. Items marked with a star are angles at which alignment lines are drawn through the endpoints of hot lines. The `measure` button measures the angle indicated by the most recent three points selected with the caret.

`parallels`	Items marked with a star are distances at which alignment lines are drawn parallel to hot lines. The `measure` button measures the distance between the most recent two points selected with the caret. Distances are nominally in inches, but the program believes the display pitch to be 100 pixels per inch.
`circles`	Items marked with a star are radii of alignment circles drawn with centers at hot points. The `measure` button measures the distance between the most recent two points selected with the caret.
`grid`	Items in this menu activate a rectangular grid of gravitating points. They have labels like `0,0+.1,.1`. The first pair of numbers is the coordinate of a point on the grid; the other pair is the x and y displacements of other points. The `off` item disables the grid (the default situation). `Measure` creates a custom-measured grid using the last two positions of the caret as diagonally adjacent grid-points.
`gravity`	The starred entry on this menu is the maximum distance that the caret will move from the mouse cursor to snap to an item on the screen.
`heating`	The `heat new` button toggles whether objects are automatically heated when created or modified. The item is marked with a '*' if set, as it is initially.

SEE ALSO

tweak(1)

BUGS

Only works on 2-bit displays (gnot, nextstation.) Doesn't compute intersections of splines with circles or arcs. No filled regions, line styles or arrowheads. Doesn't save construction lines in files. Because it draws in xor mode when rubber-banding, lines can momentarily disappear. Tracks slowly in large drawings.

NAME

ascii, unicode − interpret ASCII, Unicode characters

SYNOPSIS

`ascii` [`-8`] [`-oxdb`*n*] [`-nct`] [*text*]

`unicode` *hexmin-hexmax*

`unicode` [`-t`] *hex* [...]

`unicode` [`-n`] *characters*

`look` *hex* `/lib/unicode`

DESCRIPTION

Ascii prints the ASCII values corresponding to characters and *vice versa*; under the `-8` option, the ISO Latin-1 extensions (codes 0200-0377) are included. The values are interpreted in a settable numeric base; `-o` specifies octal, `-d` decimal, `-x` hexadecimal (the default), and `-b`*n* base *n*.

With no arguments, *ascii* prints a table of the character set in the specified base. Characters of *text* are converted to their ASCII values, one per line. If, however, the first *text* argument is a valid number in the specified base, conversion goes the opposite way. Control characters are printed as two- or three-character mnemonics. Other options are:

`-n`	Force numeric output.
`-c`	Force character output.
`-t`	Convert from numbers to running text; do not interpret control characters or insert newlines.

Unicode is similar; it converts between UTF and Unicode (see *utf*(6)). If given a range of hexadecimal numbers, *unicode* prints a table of the specified Unicode characters — their values and UTF representations. Otherwise it translates from UTF to numeric value or vice versa, depending on the appearance of the supplied text; the `-n` option forces numeric output to avoid ambiguity with numeric characters. If converting to UTF , the characters are printed one per line unless the `-t` flag is set, in which case the output is a single string containing only the specified characters. Unlike *ascii*, *unicode* treats no characters specially.

The output of *ascii* and *unicode* may be unhelpful if the characters printed are not available in the current font.

The file `/lib/unicode` contains a table of characters and descriptions, sorted in hexadecimal order, suitable for *look*(1) on the lowercase *hex* values of characters.

EXAMPLES

`ascii -d`

Print the ASCII table base 10.

`unicode p`

Print the hex value of ‘p’.

`unicode 2300`

Show which character is hex 2300.

`unicode 2300-232c`

Print a table of miscellaneous technical characters.

`look 039 /lib/unicode`

See the start of the Greek alphabet’s encoding in Unicode.

FILES

`/lib/unicode`	table of characters and descriptions.

SEE ALSO

tcs(1), *utf*(6), *font*(6)

NAME

awk − pattern-directed scanning and processing language

SYNOPSIS

`awk` [`-F`*fs*] [`-v` *var*=*value*] [*prog*] [*file ...*]

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified literally in *prog* or in one or more files specified as `-f` *file*. With each pattern there can be an associated action that will be performed when a line of a *file* matches the pattern. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern. The file name `-` means the standard input. Any *file* of the form *var*=*value* is treated as an assignment, not a file name, and is executed at the time it would have been opened if it were a file name. The option `-v` followed by *var*=*value* is an assignment to be done before *prog* is executed; any number of `-v` options may be present.

An input line is made up of fields separated by white space, or by regular expression `FS`. The fields are denoted `$1`, `$2`, ..., while `$0` refers to the entire line.

A pattern-action statement has the form

pattern `{` *action* `}`

A missing `{` *action* `}` means print the line; a missing pattern always matches. Pattern-action statements are separated by newlines or semicolons.

An action is a sequence of statements. A statement can be one of the following:

`if(` *expression* `)` *statement* [`else` *statement*]
`while(` *expression* `)` *statement*
`for(` *expression* `;` *expression* `;` *expression* `)` *statement*
`for(` *var* `in` *array* `)` *statement*
`do` *statement* `while(` *expression* `)`
`break`
`continue`
`{` [*statement ...*] `}`
expression # commonly *var* = *expression*
`print` [*expression-list*] [> *expression*]
`printf` *format* [`,` *expression-list*] [> *expression*]
`return` [*expression*]
`next` # skip remaining patterns on this input line
`delete` *array*`[` *expression* `]` # delete an array element
`exit` [*expression*] # exit immediately; status is *expression*

Statements are terminated by semicolons, newlines or right braces. An empty *expression-list* stands for `$0`. String constants are quoted `" "`, with the usual C escapes recognized within. Expressions take on string or numeric values as appropriate, and are built using the operators `+ - * / % ^` (exponentiation), and concatenation (indicated by white space). The operators `! ++ -- += -= *= /= %= ^= > >= < <= == != && || ?:` are also available in expressions. Variables may be scalars, array elements (denoted *x*`[`*i*`]`*)* or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. Multiple subscripts such as `[i,j,k]` are permitted; the constituents are concatenated, separated by the value of `SUBSEP`.

The `print` statement prints its arguments on the standard output (or on a file if >*file* or >>*file* is present or on a pipe if | *cmd* is present), separated by the current output field separator, and terminated by the output record separator. *file* and *cmd* may be literal names or parenthesized expressions; identical string values in different statements denote the same open file. The `printf` statement formats its expression list according to the format (see *fprintf*(2)). The built-in function `close(`*expr*`)` closes the file or pipe *expr*.

The mathematical functions `exp`, `log`, `sqrt`, `sin`, `cos`, and `atan2` are built in. Other built-in functions:

`length`	the length of its argument taken as a string, or of `$0` if no argument.
`rand`	random number on (0,1)
`srand`	sets seed for `rand` and returns the previous seed.
`int`	truncates to an integer value
`substr(`*s*`,` *m*`,` *n*`)`	the *n*-character substring of *s* that begins at position *m* counted from 1.
`index(`*s*`,` *t*`)`	the position in *s* where the string *t* occurs, or 0 if it does not.
`match(`*s*`,` *r*`)`	the position in *s* where the regular expression *r* occurs, or 0 if it does not. The variables `RSTART` and `RLENGTH` are set to the position and length of the matched string.
`split(`*s*`,` *a*`,` *fs*`)`	splits the string *s* into array elements *a*`[1]`, *a*`[2]`, ..., *a*`[`*n*`]`, and returns *n*. The separation is done with the regular expression *fs* or with the field separator `FS` if *fs* is not given.
`sub(`*r*`,` *t*`,` *s*`)`	substitutes *t* for the first occurrence of the regular expression *r* in the string *s*. If *s* is not given, `$0` is used.
`gsub`	same as `sub` except that all occurrences of the regular expression are replaced; `sub` and `gsub` return the number of replacements.
`sprintf(`*fmt*`,` *expr*`,` ...`)`	the string resulting from formatting *expr* ... according to the *printf* format *fmt*
`system(`*cmd*`)`	executes *cmd* and returns its exit status

The ‘‘function’’ `getline` sets `$0` to the next input record from the current input file; `getline <`*file* sets `$0` to the next record from *file*. `getline` *x* sets variable *x* instead. Finally, *cmd* `|` `getline` pipes the output of *cmd* into `getline`; each call of `getline` returns the next line of output from *cmd*. In all cases, `getline` returns 1 for a successful input, 0 for end of file, and −1 for an error.

Patterns are arbitrary Boolean combinations (with `! || &&`) of regular expressions and relational expressions. Regular expressions are as in *regexp*(6). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions, using the operators `~` and `!~`. `/`*re*`/` is a constant regular expression; any string (constant or variable) may be used as a regular expression, except in the position of an isolated regular expression in a pattern.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines from an occurrence of the first pattern though an occurrence of the second.

A relational expression is one of the following:

> *expression matchop regular-expression*
> *expression relop expression*
> *expression* `in` *array-name*
> `(`*expr*`,` *expr,...*`)` `in` *array-name*

where a relop is any of the six relational operators in C, and a matchop is either `~` (matches) or `!~` (does not match). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns `BEGIN` and `END` may be used to capture control before the first input line is read and after the last. `BEGIN` and `END` do not combine with other patterns.

Variable names with special meanings:

`FS`	regular expression used to separate fields; also settable by option `-F`*fs*.
`NF`	number of fields in the current record
`NR`	ordinal number of the current record
`FNR`	ordinal number of the current record in the current file

`FILENAME`	the name of the current input file
`RS`	input record separator (default newline)
`OFS`	output field separator (default blank)
`ORS`	output record separator (default newline)
`OFMT`	output format for numbers (default `%.6g`)
`SUBSEP`	separates multiple subscripts (default 034)
`ARGC`	argument count, assignable
`ARGV`	argument array, assignable; non-null members are taken as file names
`ENVIRON`	array of environment variables; subscripts are names.

Functions may be defined (at the position of a pattern-action statement) thus:

```
function foo(a, b, c) { ...; return x }
```

Parameters are passed by value if scalar and by reference if array name; functions may be called recursively. Parameters are local to the function; all other variables are global. Thus local variables may be created by providing excess parameters in the function definition.

EXAMPLES

```
length > 72
```

Print lines longer than 72 characters.

```
{ print $2, $1 }
```

Print first two fields in opposite order.

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```

Same, with input fields separated by comma and/or blanks and tabs.

```
      { s += $1 }
END   { print "sum is", s, " average is", s/NR }
```

Add up first column, print sum and average.

```
/start/, /stop/
```

Print all lines between start/stop pairs.

```
BEGIN {     # Simulate echo(1)
      for (i = 1; i < ARGC; i++) printf "%s ", ARGV[i]
      printf "\n"
      exit }
```

SEE ALSO

sed(1), *regexp*(6),
A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The AWK Programming Language,* Addison-Wesley, 1988.

BUGS

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate `""` to it.
The scope rules for variables in functions are a botch; the syntax is worse.

NAME

basename − strip file name affixes

SYNOPSIS

`basename` *string* [*suffix*]

DESCRIPTION

Basename deletes any prefix ending in `/` and the *suffix,* if present in *string,* from *string,* and prints the result on the standard output.

NAME

bc − arbitrary-precision arithmetic language

SYNOPSIS

`bc` [`-c`] [`-l`] [*file* ...]

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides arithmetic on numbers of arbitrary length with up to 100 digits right of the decimal point. It takes input from any files given, then reads the standard input. The `-l` argument stands for the name of an arbitrary precision math library. The following syntax for *bc* programs is like that of C; *L* means letter `a-z`, *E* means expression, *S* means statement.

Lexical

comments are enclosed in `/* */`
newlines end statements

Names

simple variables: *L*
array elements: *L* `[` *E* `]`
The words `ibase`, `obase`, and `scale`

Other operands arbitrarily long numbers with optional sign and decimal point.
`(` *E* `)`
`sqrt (` *E* `)`
`length (` *E* `)`
  number of significant decimal digits
`scale (` *E* `)`
  number of digits right of decimal point
L `(` *E* `,` ... `,` *E* `)`
  function call

Operators

`+ - * / % ^` (`%` is remainder; `^` is power)
`++ --`
`== <= >= != < >`
`= += -= *= /= %= ^=`

Statements

E
`{` *S* `;` ... `;` *S* `}`
`print` *E*
`if (` *E* `)` *S*
`while (` *E* `)` *S*
`for (` *E* `;` *E* `;` *E* `)` *S*
null statement
`break`
`quit`
`"`text`"`

Function definitions

`define` *L* `(` *L* `,` ... `,` *L* `){`
`auto` *L* `,` ... `,` *L*
S `;` ... `;` *S*
`return` *E*
`}`

Functions in `-l` math library
`s(`*x*`)` sine
`c(`*x*`)` cosine

`e(`*x*`)`	exponential
`l(`*x*`)`	log
`a(`*x*`)`	arctangent
`j(`*n, x*`)`	Bessel function

All function arguments are passed by value.

The value of an expression at the top level is printed unless the main operator is an assignment. Text in quotes, which may include newlines, is also printed. Either semicolons or newlines may separate statements. Assignment to `scale` influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to `ibase` or `obase` set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. Automatic variables are pushed down during function calls. In a declaration of an array as a function argument or automatic variable empty square brackets must follow the array name.

Bc is actually a preprocessor for *dc*(1), which it invokes automatically, unless the `-c` (compile only) option is present. In this case the *dc* input is sent to the standard output instead.

EXAMPLES

Define a function to compute an approximate value of the exponential. Use it to print 10 values. (The exponential function in the library gives better answers.)

```
scale = 20
define e(x) {
      auto a, b, c, i, s
      a = 1
      b = 1
      s = 1
      for(i=1; 1; i++) {
            a *= x
            b *= i
            c = a/b
            if(c == 0) return s
            s += c
      }
}
for(i=1; i<=10; i++) print e(i)
```

FILES

`/sys/lib/bclib` mathematical library

SEE ALSO

dc(1), *hoc*(1)

BUGS

No `&&`, `||`, or `!` operators.
A `for` statement must have all three `E`s.
A `quit` is interpreted when read, not when executed.

NAME

bind, mount, unmount − change name space

SYNOPSIS

`bind` [*option ...*] *new old*

`mount` [*option ...*] [`-t`] [`-s` *server*] *servename old* [*spec*]

`unmount` [*new*] *old*

DESCRIPTION

Bind and *mount* modify the file name space of the current process and other processes in the same name space group (see *fork*(2)). For both calls, *old* is the name of an existing file or directory in the current name space where the modification is to be made.

For *bind*, *new* is the name of another (or possibly the same) existing file or directory in the current name space. After a successful *bind*, the file name *old* is an alias for the object originally named by *new*; if the modification doesn't hide it, *new* will also still refer to its original file. The evaluation of *new* (see *intro*(2)) happens at the time of the *bind*, not when the binding is later used.

The *servename* argument to *mount* is the name of a file that, when opened, yields an existing connection to a file server. Almost always, *servename* will be a file in `/srv` (see *srv*(3)). In the discussion below, *new* refers to the file named by the *new* argument to *bind* or the root directory of the service available in *servename* after a *mount.* Either both *old* and *new* files must be directories, or both must not be directories.

Options control aspects of the modification to the name space:

(none)	Replace the *old* file by the new one. Henceforth, an evaluation of *old* will be translated to the new file. If they are directories (for *mount*, this condition is true by definition), *old* becomes a *union directory* consisting of one directory (the new file).
`-b`	Both files must be directories. Add the new directory to the beginning of the union directory represented by the old file.
`-a`	Both files must be directories. Add the new directory to the end of the union directory represented by the old file.
`-c`	This can be used in addition to any of the above to permit creation in a union directory. When a new file is created in a union directory, it is placed in the first element of the union that permits creation.

Mount takes options to specify the level of authentication to perform on the connection. Option `-s` *server* authenticates the user and checks that the machine answering the request is indeed the desired *server.* Option `-t` is like `-s` but trusts that the (unspecified) server is the right machine. If neither `-s` nor `-t` is specified, no authentication is performed, but most file servers will reject the *mount*.

The *spec* argument to *mount* is passed in the *attach*(5) message to the server, and selects among different file trees served by the server.

The *srv*(3) service registry device, normally bound to `/srv`, is a convenient rendezvous point for services that can be mounted. After bootstrap, the file `/srv/boot` contains the communications port to the file system from which the system was loaded.

The effects of *bind* and *mount* can be undone with the *unmount* command. If two arguments are given to *unmount*, the effect is to undo a *bind* or *mount* with the same arguments. If only one argument is given, everything bound to or mounted upon *new* is unmounted.

EXAMPLES

To compile a program with the C library from July 16, 1992:

```
mount -t /srv/boot /n/dump dump
bind /n/dump/1992/0716/mips/lib/libc.a /mips/lib/libc.a
mk
```

SEE ALSO

bind(2), *open*(2), *srv*(3), *srv*(4)

NAME

bundle − collect files for distribution

SYNOPSIS

`bundle` *file ...*

DESCRIPTION

Bundle writes on its standard output a shell script for *rc*(1) or a Bourne shell which, when executed, will recreate the original *files*. Its main use is for distributing small numbers of text files by *mail*(1).

Although less refined than standard archives from *ar*(1) or *tar*(1), a *bundle* file is self-documenting and complete; little preparation is required on the receiving machine.

EXAMPLES

```
bundle mkfile *.[ch] | mail kremvax!boris
```

Send a makefile to Boris together with related `.c` and `.h` files. Upon receiving the mail, Boris may save the file sans postmark, say in `gift/horse`, then do

```
cd gift; rc horse; mk
```

SEE ALSO

ar(1), *tar*(1), *mail*(1)

BUGS

Bundle will not create directories and is unsatisfactory for non-text files.
Beware of gift horses.

NAME

c++/2c, c++/kc, c++/vc, c++/8c, c++/zc, c++/2l, c++/kl, c++/vl, c++/8l, c++/zl − C++ compilers and loaders

SYNOPSIS

`c++/2c` [*options*] *name ...*
`c++/8c` [*options*] *name ...*
`c++/kc` [*options*] *name ...*
`c++/vc` [*options*] *name ...*
`c++/zc` [*options*] *name ...*
`c++/2l` [*options*] *name ...*
`c++/8l` [*options*] *name ...*
`c++/kl` [*options*] *name ...*
`c++/vl` [*options*] *name ...*
`c++/zl` [*options*] *name ...*

DESCRIPTION

The c++ compilers, *c++/?c,* compile the named C++ files into object files for the specified architecture (see *2c*(1)). They use *cpp*(1) as the preprocessor, *cfront* 3.0.1 as the C++ to C translator, and the appropriate C compiler such as *2c*. The c++ loaders, *C++/?l*, load object files using appropriate object loaders (see *2l*(1)) and `patch`, the C++ static constructor initializer.

The compilers and loaders use C++ and APE (ANSI C/POSIX) include files and libraries.

Let the first letter of the base name of the compiler or loader be *O*= `2`, `8`, `k`, `v`, or `z`.

The compiler options are:

`-d`	Don’t expand inline functions.
`-o` *obj*	Place output in file *obj* (allowed only if there is just one input file). Default is to take the last element of the input path name, strip any trailing `.c`, and append `.`*O*.
`-v`	Print the version number of the compiler and the commands as they are executed. A second `-v` causes the commands that would be executed to be printed without actually executing them.
`-w`	Print warning messages.
`-x file`	Take cross compiling information from file. By default, this information is taken from `/sys/lib/c++/`*O*`.sz`.
`-A`	Complain about functions used without a new-style ANSI function prototype.
`-B`	Turn off the action of the `-A` flag. This option is on by default.
`-D`*name*=*def* `-D`*name*	Define the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as `1`.
`-E`	Print the preprocessed version of the file on standard output.
`-F`	Print the preprocessed and *cfronted* version of the file on standard output.
`-I`*dir*	`#include` files whose names do not begin with `/` are always sought first in the directory of the *file* argument, then in directories named in `-I` options, then in `/$objtype/include/c++`, `/sys/include/c++`, `/$objtype/include/ape`, and `/sys/include/ape`.
`-O`	Perform object code optimization. This option is on by default.
`-N`	Turn off the action of the `-O` flag.
`-S`	Print an assembly language version of the object code on standard output.

`-U`*name*	Remove any initial definition of *name*.

The loader options are:

`-o` *out*	Place output in file *out*. Default is *O*`.out`.

EXAMPLE

To produce a MIPS executable `prog` from C++ files `main.c`, `sub.c`, and using the task library:

```
c++/vc main.c sub.c
c++/vl -o prog main.v sub.v m.v -ltask
```

FILES

`/sys/include/c++`	directory for machine-independent `#include` directives.
`/sys/include/ape`	directory for machine-independent `#include` directives.
`/$objtype/include/c++`	directory for machine-dependent `#include` directives.
`/$objtype/include/ape`	directory for machine-dependent `#include` directives.
`/$objtype/lib/c++`	C++ libraries.
`/$objtype/lib/ape/libap.a`	ANSI C/POSIX library.
`/sys/lib/c++/`*O*`.sz`	Cross-compilation information for `cfront`.
`/$cputype/bin/c++/cfront`	C++ to C translator.
`/$cputype/bin/c++/patch`	C++ static constructor initializer.

SEE ALSO

2c(1), *2a*(1), *2l*(1), *db*(1) *cpp*(1), *mk*(1), *nm*(1), *pcc*(1) *rl*(1),

BUGS

The task library works only for the MIPS and the SPARC; it will not work for the other machines any time soon. The Interrupt class is not yet supported.

NAME

cal − print calendar

SYNOPSIS

`cal` [*month*] [*year*]

DESCRIPTION

Cal prints a calendar. *Month* is either a number between 1 and 12, a lower case month name, or a lower case three-letter prefix of a month name. *Year* can be between 1 and 9999. If either *month* or *year* is omitted, the current month or year is used. If only one argument is given, and it is a number larger than 12, a calendar for all twelve months of the given year is produced; otherwise a calendar for just one month is printed. The calendar produced is that for England and her colonies.

Try `cal september 1752`.

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that `cal 90` refers to the early Christian era, not the 20th century.

NAME

cat, read − catenate files

SYNOPSIS

`cat` [*file* ...]
`read` [*file*]

DESCRIPTION

Cat reads each *file* in sequence and writes it on the standard output. Thus

```
cat file
```

prints a file and

```
cat file1 file2 >file3
```

concatenates the first two files and places the result on the third.

If no *file* is given, *cat* reads from the standard input. Output is buffered in blocks matching the input.

Read copies one line from the named file to standard output. It is useful in interactive *rc*(1) scripts.

SEE ALSO

cp(1)

DIAGNOSTICS

Read exits with status `eof` on end of file.

BUGS

Beware of `cat a b >a` and `cat a b >b`, which destroy input files before reading them.

NAME

chgrp − change file group

SYNOPSIS

`chgrp` *group file ...*

DESCRIPTION

The group of each named file is changed to *group,* which should be a name known to the server holding the file.

A file's group can be changed by the file's owner, if the owner is a member of the new group, or by the leader of both the file's current group and the new group.

SEE ALSO

ls(1), *chmod*(1), *stat*(2)

NAME

chmod − change mode

SYNOPSIS

`chmod` *mode file ...*

DESCRIPTION

The mode of each named file is changed according to *mode,* which may be an octal number or a symbolic change to the existing mode. A *mode* is an octal number constructed from the OR of the following modes.

0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic *mode* has the form:

> [who] *op permission*

The *who* part is a combination of the letters `u` (for user's permissions), `g` (group) and `o` (other). The letter `a` stands for `ugo`. If *who* is omitted, the default is `a`.

Op can be + to add *permission* to the file's mode, − to take away *permission* and = to assign *permission* absolutely (all other bits will be reset).

Permission is any combination of the letters `r` (read), `w` (write), `x` (execute), `a` (append only), and `l` (exclusive access).

Only the owner of a file or the group leader of its group may change the file's mode.

SEE ALSO

ls(1), *stat*(2), *stat*(5)

NAME

cmp − compare two files

SYNOPSIS

`cmp` [`-lsL`] *file1 file2* [*offset1* [*offset2*]]

DESCRIPTION

The two files are compared. A diagnostic results if the contents differ, otherwise there is no output.

The options are:

`l` Print the byte number (decimal) and the differing bytes (hexadecimal) for each difference.

`s` Print nothing for differing files, but set the exit status.

`L` Print the line number of the first differing byte.

If offsets are given, comparison starts at the designated byte position of the corresponding file. Offsets that begin with `0x` are hexadecimal; with `0`, octal; with anything else, decimal.

SEE ALSO

diff(1)

DIAGNOSTICS

If a file is inaccessible or missing, the exit status is `open`. If the files are the same, the exit status is `empty`. If they are the same except that one is longer than the other, the exit status is `EOF`. Otherwise *cmp* reports the position of the first disagreeing byte and the exit status is `differ`.

NAME

comm − select or reject lines common to two sorted files

SYNOPSIS

`comm` [`-123`] *file1 file2*

DESCRIPTION

Comm reads *file1* and *file2,* which are ordered in ASCII collating sequence, and produces a three column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name − means the standard input.

Flag `1`, `2`, or `3` suppresses printing of the corresponding column.

EXAMPLES

```
comm -12 file1 file2
```

Print lines common to two sorted files.

```
deroff -w /mail/lib/names.last | tr a-z A-Z | sort -u >temp
spell temp | comm -13 - temp
```

Print names that are known both to *mail*(1) and *spell*(1)

SEE ALSO

sort(1), *cmp*(1), *diff*(1), *uniq*(1)

NAME

compress, uncompress − compress and expand data

SYNOPSIS

`compress` [`−dfvcV`] [`−b` *bits*] [*name ...*]

DESCRIPTION

Compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension `.Z,` while keeping the same ownership modes, access and modification times. If no files are specified, the standard input is compressed to the standard output. Compressed files can be restored to their original form using *compress* `−d`. The script, */rc/bin/uncompress,* performs the command *‘‘compress −d $*’’.*

The `−f` option will force compression of *name.* This is useful for compressing an entire directory, even if some of the files do not actually shrink. If `−f` is not given and *compress* is run in the foreground, the user is prompted as to whether an existing file should be overwritten.

The `−c` option makes *compress* write to the standard output; no files are changed.

The `−v` option prints compression statistics and the `−V` option prints the version of the program.

Compress uses the modified Lempel-Ziv algorithm popularized in "A Technique for High Performance Data Compression", Terry A. Welch, *IEEE Computer,* vol. 17, no. 6 (June 1984), pp. 8-19. Common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit specified by the `−b` flag is reached (default 16). *Bits* must be between 9 and 16. The default can be changed in the source to allow *compress* to be run on a smaller machine.

After the *bits* limit is attained, *compress* periodically checks the compression ratio. If it is increasing, *compress* continues to use the existing code dictionary. However, if the compression ratio decreases, *compress* discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next "block" of the file.

Note that the `−b` flag is omitted for *compress -d*, since the *bits* parameter specified during compression is encoded within the output, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is attempted.

The amount of compression obtained depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50−60%. Compression is generally much better than that achieved by Huffman coding (as used in *pack*), or adaptive Huffman coding (*compact*), and takes less time to compute.

Under the `−v` option, a message is printed yielding the percentage of reduction for each file compressed.

If the `−V` option is specified, the current version and compile options are printed on stderr.

BUGS

Although compressed files are compatible between machines with large memory, `−b`12 should be used for file transfer to architectures with a small process data space (64KB or less, as exhibited by the DEC PDP series, the Intel 80286, etc.)

NAME

con, telnet, hayes, cu, rx, xms, xmr − remote login, execution, and XMODEM file transfer

SYNOPSIS

`con` [`-dCrv`] [`-l` [*remuser*]] [`-c` *cmd*] *net*`!`*machine*

`telnet` [`-dCr`] *net*`!`*machine*

`hayes` [`-p`] *number* [*device*]

`cu` *number*

`rx` [`-n`] *net*`!`*machine* [*command-word ...*]

`xms` *file*

`xmr` *file*

DESCRIPTION

Con connects to the computer whose network address is *net*`!`*machine* and logs in if possible. With no options, the account name used on the remote system is the same as that on the local system. Standard input and output go to the local machine.

Options are:

`-l`	with an argument causes *remuser* to be used as the account name on the remote system. Without an argument this option disables automatic login and a normal login session ensues.
`-C`	forces cooked mode, that is, local echo.
`-c`	runs *cmd* as if it had been typed as a command from the escape mode. This is used by *cu*.
`-v`	(verbose mode) causes information about connection attempts to be output to standard error. This can be useful when trying to debug network connectivity.
`-d`	causes debugging information to be output to standard error.
`-r`	suppresses printing of any carriage return followed by a new line. This is useful since carriage return is a printable character in Plan 9.

The control-\ character is a local escape. It prompts with the local machine name and >>>. Legitimate responses to the prompt are

`i`	Send a quit [sic] signal to the remote machine.
`q`	Exit.
`b`	Send a break.
`.`	Return from the escape.
`!cmd`	Run the command with the network connection as its standard input and standard output. Standard error will go to the screen. This is useful for transmitting and receiving files over the connections using programs such as *xms*.

Telnet is similar to con, but it uses the *telnet* protocol to communicate with the remote machine.

Hayes dials a number using the Hayes modem protocol. Option `p` uses pulse dialing rather than the default tone dialing. If specified, *device* is the file opened for the call. The default is `/dev/eia0`.

Cu is a shell script that uses *hayes* and *con* to connect to a machine via a modem. If the machine is equipped with a local modem, it is used. Otherwise, the call is placed through Datakit.

Rx executes one shell command on the remote machine as if logged in there, but with local standard input and output. Unquoted shell metacharacters in the command are interpreted locally, quoted ones remotely. The assignment `REXEC=1` appears in the remote environment.

Network addresses for both *con* and *rx* have the form *network* ! *host*. Supported networks are those listed in `/net`.

The commands *xms* and *xmr* respectively send and receive a single file using the XMODEM protocol. They use standard input and standard output for communication and are intended for use with *con*.

EXAMPLES

```
rx kremvax cat file1 >file2
```

Copy remote *file1* to local *file2*.

```
rx kremvax cat file1 '>file2'
```

Copy remote *file1* to remote *file2*.

```
eqn paper | rx kremvax troff -ms | rx deepthought lp
```

Parallel processing: do each stage of a pipeline on a different machine.

SEE ALSO

push(1)

BUGS

Under *rx*, a program that should behave specially towards terminals may not: e.g., remote shells will not prompt. Also under *rx*, the remote standard error and standard output are combined and go inseparably to the local standard output.

NAME

cp, mv, rename − copy, move files

SYNOPSIS

`cp` [`-z` [*bufsize*]] *file1 file2*
`cp` [`-z` [*bufsize*]] *file ... directory*

`mv` *file1 file2*
`mv` *file ... directory*

DESCRIPTION

In the first form *file1* is any name and *file2* is any name except an existing directory. In the second form the commands copy or move one or more *files* into a *directory* under their original file names, as if by a sequence of commands in the first form. Thus `cp f1 f2 dir` is equivalent to `cp f1 dir/f1; cp f2 dir/f2`.

Cp copies the contents of plain *file1* to *file2*. The mode and owner of *file2* are preserved if it already exists; the mode of *file1* is used otherwise. The `-z` option says to preserve 'holes'; see *seek*(2). The reads and writes are done in hunks of size *bufsize*.

Mv moves *file1* to *file2*. If the files are in the same directory, *file1* is just renamed; otherwise *mv* behaves like *cp*. *Mv* will rename directories, but it refuses to move a directory into another directory.

SEE ALSO

cat(1), *stat*(2), *push*(1)

DIAGNOSTICS

Cp and *mv* refuse to copy or move files onto themselves.

NAME

cpp − C language preprocessor

SYNOPSIS

`cpp` [*option ...*] [*ifile* [*ofile*]]

DESCRIPTION

Cpp interprets ANSI C preprocessor directives and does macro substitution. The input *ifile* and output *ofile* default to standard input and standard output respectively.

The options are:

`-U`*name*
`-D`*name*
`-D`*name*`=`*def*
`-I`*dir* Same as in *2c*(1).
`-M` Generate no output except a list of include files. Use twice to list files in angle brackets.
`-N` Turn off default include directories. All must be specified with `-I`. Without this option, `/$objtype/include` and `/sys/include` are used as the last two searched directories for include directives, where `$objtype` is read from the environment.
`-V` Print extra debugging information.
`-+` Understand C++ comments.

The output file contains processed text sprinkled with lines that show the original input line numbering:

`#line` *linenumber* `"`*ifile*`"`

The input language is as described in the ANSI C standard. The C compilers do not use *cpp;* they contain their own simple but adequate preprocessor, so *cpp* is usually superfluous.

FILES

`/sys/include`	directory for machine-independent include files
`/$objtype/include`	directory for machine-dependent include files

SEE ALSO

2c(1)

NAME

cpu − connection to cpu server

SYNOPSIS

`cpu` [`-h` *server*] [`-c` *cmd args ...*]

DESCRIPTION

Cpu starts an *rc*(1) running on the *server* machine, or the machine named in the `$cpu` environment variable if there is no `-h` option. *Rc*'s standard input, output, and error files will be `/dev/cons` in the name space where the *cpu* command was invoked. Normally, *cpu* is run in an *8½*(1) window on a terminal, so *rc* output goes to that window, and input comes from the keyboard when that window is current. *Rc*'s current directory is the working directory of the *cpu* command itself.

The name space for the new *rc* is an analogue of the name space where the *cpu* command was invoked: it is the same except for architecture-dependent bindings such as `/bin` and the use of fast paths to file servers, if available.

If a `-c` argument is present, the remainder of the command line is executed by *rc* on the server, and then *cpu* exits.

The name space is built by running `/usr/$user/lib/profile` with the root of the invoking name space bound to `/mnt/term`. The `service` environment variable is set to `cpu`; the `cputype` and `objtype` environment variables reflect the server's architecture.

FILES

The name space of the terminal side of the `cpu` command is mounted on the CPU side on directory `/mnt/term`.

SEE ALSO

rc(1), *8½*(1)

BUGS

Binds and mounts done after the terminal `lib/profile` is run are not reflected in the new name space.

NAME

date − print the date

SYNOPSIS

`date` [*option*] [*seconds*]

DESCRIPTION

Print the date, in the format

```
Tue Aug 16 17:03:52 CDT 1977
```

The options are

`−u` Report Greenwich Mean Time (GMT) rather than local time.

`−n` Report the date as the number of seconds since the epoch, 00:00:00 local time, January 1, 1970.

The conversion from Greenwich Mean Time to local time depends on the `$timezone` environment variable; see *ctime*(2).

If the optional argument *seconds* is present, it is used as the time to convert rather than the real time.

FILES

`/env/timezone`	Current timezone name and adjustments.
`/adm/timezone`	A directory containing timezone tables.
`/adm/timezone/local`	Default timezone file, copied by *init*(8) into `/env/timezone`.

NAME

db, dbfmt − debugger

SYNOPSIS

`db` [*option ...*] [*textfile* [*memfile*]]

`db` [`-k`] *pid*

`dbfmt`

DESCRIPTION

Db is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of Plan 9 programs.

Textfile is normally an executable program file or `/proc/`*pid*`/text`. *Memfile* is the memory image of a process, usually obtained from `/proc/`*pid*`/mem`. If there is exactly one argument, and it is numeric, then it is used as a *pid* to find the `text` and `mem` files in `/proc/`*pid* **.**

Requests to *db* are read from the standard input and responses are to the standard output. The options are

`-k` *pid* Use the kernel for the *textfile* and *memfile*, with the kernel stack of process *pid*.

`-w` Create *textfile* and *memfile* if they don't exist; open them for writing as well as reading.

`-I`*path* Directory in which to look for relative path names in `$<` and `$<<` commands.

`-m`*machine*
Assume instructions are for the given CPU type (one of `386`, `68020`, `960`, `hobbit`, `mips`, `mipsco`, `sparc`, or `sunsparc`) instead of using the magic number in the text file to select the CPU type.

In general requests to *db* have the following form. Multiple requests on one line must be separated by `;`.

[*address*] [`,` *count*] [*command*]

If *address* is present then the current position, called 'dot', is set to *address*. Initially dot is set to 0. In general commands are repeated *count* times. Dot advances between repetitions. The default *count* is 1. *Address* and *count* are expressions.

Expressions

Expressions are evaluated as long ints.

`.` The value of dot.

`+` The value of dot incremented by the current increment.

`^` The value of dot decremented by the current increment.

`"` The last *address* typed.

integer A number, in decimal radix by default. The prefixes `0` and `0o` and `0O` (zero oh) force interpretation in octal radix; the prefixes `0t` and `0T` force interpretation in decimal radix; the prefixes `0x`, `0X`, and `#` force interpretation in hexadecimal radix. Thus `020`, `0o20`, `0t16`, and `#10` all represent sixteen.

integer`.`*fraction*
A single-precision floating point number.

`'` *c* `'` The Unicode value of a character. `\` may be used to escape a `'`.

`<`*name* The value of *name*, which is either a variable name or a register name. *db* maintains a number of variables named by single letters or digits. The register names are those printed by the `$r` command.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. `\` may be used to escape other characters. The location of the *symbol* is calculated from the symbol table in *textfile*.

routine **.** *name*
The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*; if *routine* is omitted, the active procedure is assumed.

file **:** *integer*
The address of the instruction corresponding to the C source statement at the indicated line number of the file. If the source line contains no executable statement, the address of the instruction associated with the nearest executable source line is returned. Files begin at line 1. If multiple files of the same name are loaded, an expression of this form resolves to the first file encountered in the symbol table.

`(` *exp* `)` The value of the expression *exp.*

Monadic operators

`*`*exp* The contents of the location addressed by *exp* in *memfile*.

`@`*exp* The contents of the location addressed by *exp* in *textfile*.

`-`*exp* Integer negation.

`~`*exp* Bitwise complement.

`%`*exp* If *exp* is used as an address, it is in register space; see ‘Addresses’.

Dyadic operators are left associative and are less binding than monadic operators.

e1`+`*e2* Integer addition.

e1`-`*e2* Integer subtraction.

e1`*`*e2* Integer multiplication.

e1`%`*e2* Integer division.

e1`&`*e2* Bitwise conjunction.

e1`|`*e2* Bitwise disjunction.

e1`#`*e2* *E1* rounded up to the next multiple of *e2*.

Commands

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands `?` and `/` may be followed by `*`; see ‘Addresses’ for further details.)

`?`*f* Locations starting at *address* in *textfile* are printed according to the format *f*.

`/`*f* Locations starting at *address* in *memfile* are printed according to the format *f.*

`=`*f* The value of *address* itself is printed in the styles indicated by the format *f*.

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. If no format is given then the last format is used.

Most format letters fetch some data, print it, and advance (a local copy of) dot by the number of bytes fetched. The total number of bytes in a format becomes the *current increment.*

`o`	Print two-byte integer in octal.
`O`	Print four-byte integer in octal.
`q`	Print two-byte in signed octal.
`Q`	Print four-byte in signed octal.
`d`	Print two-byte in decimal.
`D`	Print four-byte in decimal.
`x`	Print two-byte in hexadecimal.

`X`	Print four-byte in hexadecimal.
`u`	Print two-byte in unsigned decimal.
`U`	Print four-byte in unsigned decimal.
`f`	Print as a single-precision floating point number.
`F`	Print double-precision floating point.
`b`	Print the addressed byte in hexadecimal.
`c`	Print the addressed byte as an ASCII character.
`C`	Print the addressed byte as a character. Printable ASCII characters are represented normally; others are printed in the form `\xnn`.
`s`	Print the addressed characters, as a UTF string, until a zero byte is reached. Advance dot by the length of the string, including the zero terminator.
`S`	Print a string using the escape convention (see `C` above).
`r`	Print the addressed two-byte integer as a rune.
`R`	Print the addressed two-byte integers as runes until a zero rune is reached. Advance dot by the length of the string, including the zero terminator.
`Y`	Print a four-byte integer in date format (see *ctime*(2)).
`i`	Print as machine instructions. This style of printing causes variables 0, (1, ...) to be set to the offset parts of the first (second, ...) operand of the instruction.
`I`	As `i` above, but print the machine instructions in an alternate form if possible: `sunsparc` and `mipsco` reproduce the manufacturers' syntax.
`M`	Print the addressed machine instruction in a machine dependent hexadecimal form.
`a`	Print the value of dot in symbolic form. Dot is unaffected.
`z`	Print the function name, source file, and line number corresponding to dot (textfile only). Dot is unaffected.
`p`	Print the addressed value in symbolic form. Dot is advanced by the size of a machine address.
`t`	When preceded by an integer tabs to the next appropriate tab stop. For example, `8t` moves to the next 8-space tab stop. Dot is unaffected.
`n`	Print a newline. Dot is unaffected.
`"..."`	Print the enclosed string. Dot is unaffected.
`^`	Dot is decremented by the current increment. Nothing is printed.
`+`	Dot is incremented by 1. Nothing is printed.
`-`	Dot is decremented by 1. Nothing is printed.

newline Update dot by the current increment. Repeat the previous command with a *count* of 1.

`[?/]l` *value mask*

Words starting at dot are masked with *mask* and compared with *value* until a match is found. If `l` is used, the match is for a two-byte integer; `L` matches four bytes. If no match is found then dot is unchanged; otherwise dot is set to the matched location. If *mask* is omitted then ~0 is used.

`[?/]w` *value ...*

Write the two-byte *value* into the addressed location. If the command is `W`, write four bytes.

`[?/]m` *s b e f* `[?]`

New values for (*b, e, f*) in the map entry named *s* are recorded. Valid map entry names are *text, data, ublock,* or *regs*. If less than three address expressions are given then the remaining map parameters are left unchanged. The address type (instruction or data) is unchanged in any case. If the list is terminated by `?` or `/` then the file (*textfile* or *memfile* respectively) is used for subsequent requests. For example, `/m?` causes `/` to refer to *textfile*.

`>`*name* Dot is assigned to the variable or register named.

`!` Tem rest of the line is passed to the *rc*(1) for execution.

`$`*modifier*

Miscellaneous commands. The available *modifiers* are:

<*f*	Read commands from the file *f*. If this command is executed in a file, further commands in the file are not seen. If *f* is omitted, the current input stream is terminated. If a *count* is given, and is zero, the command is ignored. The value of the count is placed in variable `9` before the first command in *f* is executed. A common use for this command is to print the fields of a structure. The *dbfmt* program takes a structure description on standard input and produces a file on standard output suitable for use in a *addr*`$<`*f* command. The `-s`*name* option of *2c*(1) produces a structure description for structure or union *name*.
<<*f*	Similar to < except it can be used in a file of commands without causing the file to be closed. Variable `9` is saved during the execution of this command, and restored when it completes. There is a (small) limit to the number of << files that can be open at once.
>*f*	Append output to the file *f*, which is created if it does not exist. If *f* is omitted, output is returned to the terminal.
`?`	Print process id, the condition which caused stopping or termination, as well as the registers. This is the default if *modifier* is omitted.
`r`	Print the general registers and the instruction addressed by `pc`. Dot is set to `pc`.
`R`	Like `$r`, but include miscellaneous registers such as the kernel stack pointer and floating point registers.
`f`	Print floating-point register values as single-precision floating point numbers.
`F`	Print floating-point register values as double-precision floating point numbers.
`b`	Print all breakpoints and their associated counts and commands.
`c`	C stack backtrace. If *address* is given then it is taken as the address of the current frame; otherwise, the current C frame pointer is used. If `C` is used then the names and (long) values of all parameters, automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.
`a`	Set the maximum number of arguments printed by `$c` or `$C` to *address*. The default is 20.
`e`	The names and values of all external variables are printed.
`w`	Set the page width for output to *address* (default 80).
`s`	Set the limit for symbol matches, used in printing addresses, to *address* (default 255).
`q`	Exit from *db*.
`v`	Print all non zero variables.
`m`	Print the address maps.
`k`	Simulate kernel memory management.
`p`	Use kernel data and stack maps for the specified process.
	`$k` and `$p` are used for system debugging (see the Examples section).
`M`*machine*	Set the *machine* type used for disassembling instructions.

: *modifier*

Manage a subprocess. Available modifiers are:

`h`	Halt an asynchronously running process to allow breakpointing. Unnecessary for processes created under *db,* e.g. by `:r`.
`b`*c*	Set breakpoint at *address*. The breakpoint is executed *count*−1 times before causing a stop. Also, if a command *c* is given it is executed at each breakpoint and if it sets dot to zero the breakpoint causes a stop.
`d`	Delete breakpoint at *address*.
`r`	Run *textfile* as a subprocess. If *address* is given the program is entered at that point; otherwise the standard entry point is used. *Count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command.
`c`*s*	The subprocess is continued. If *s* is omitted or nonzero, the subprocess is sent the note that caused it to stop. If 0 is specified, no note is sent. (If the stop was due to a breakpoint or single-step, the corresponding note is elided before continuing.) Breakpoint

skipping is the same as for `r`.

`s`*s*	As for `c` except that the subprocess is single stepped for *count* machine instructions. If a note is pending, it is received before the first instruction is executed. If there is no current subprocess then *textfile* is run as a subprocess as for `r`. In this case no note can be sent; the remainder of the line is treated as arguments to the subprocess.
`S`*s*	Identical to `s` except the subprocess is single stepped for *count* lines of C source. In optimized code, the correspondence between C source and the machine instructions is approximate at best.
`x`	The current subprocess, if any, is released by *db* and allowed to continue executing normally.
`k`	The current subprocess, if any, is terminated.
`n`*c*	Display the pending notes for the process. If *c* is specified, first delete *c'th* pending note.

Variables

db provides a number of variables. Named variables are set initially by *db* but are not used subsequently. Numbered variables are reserved for communication as follows.

`0`, `1`, ...	The offset parts of the first, second, ... operands of the last instruction printed. Meaningless if the operand was a register.
`9`	The count on the last `$<` or `$<<` command.

On entry the following are set from the system header in the *memfile*. If *memfile* does not appear to be a memory image then these values are set from *textfile*.

`b`	The base address of the data segment.
`d`	The data segment size.
`e`	The entry point.
`m`	The 'magic' number (see *a.out*(6)).
`s`	The stack segment size.
`t`	The text segment size.

Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by one or more quadruples (*t, b, e, f*), mapping an address of type *t* (text, data, user block, or registers) in the range *b* through *e* to the part of the file beginning at address *f*. An address *a* of type *t* is mapped to a file address by finding a quadruple of type *t*, for which $b \le a < e$; the file address is $address + f - b$. As a special case, if a text space address is not found, a second search is made for the same address in data space.

Typically, the text segment of a program is mapped as text space, the data and bss segments as data space. If *textfile* is an executable file or if *memfile* is a memory image, maps are set accordingly. Otherwise, a single 'data space' map is set up, with *b* and *f* set to zero, and *e* set to a huge number; thus the entire file can be examined without address translation.

The `?` and `/` commands attempt to examine text and data space respectively. `?*` tries for data space (in *textfile*); `/*` accesses text space (in *memfile*).

Registers in process and core images are a special case; they live in a special 'register' address space starting at `%0`; the layout of this space is machine-dependent. `%` addresses are mapped to the registers for the 'current frame,' set by local variable references, and reset to the outermost frame (the 'real' registers) whenever a process runs or a stack trace is requested.

Simulated memory management translations (the `$k` and `$p` commands) are done before the mapping described above.

EXAMPLES

To set a breakpoint at the beginning of `write()` in extant process 27:

```
db 27
:h
```

```
write:b
:c
```

To examine the Plan 9 kernel stack for process 27:

```
db -k 27
$C
```

Similar, but using a kernel `test`:

```
db test /proc/27/mem
27$p
$C
```

To print the fields of the `Dir` structure at address `#20000`, assuming `main.c` includes a declaration of that structure:

```
!2c -sDir -o /dev/null main.c | dbfmt > Dir.dbfmt
#20000$<Dir.dbfmt
```

To set a breakpoint at the entry of function `parse` when the local variable `argc` in `main` is equal to 1:

```
parse:b *main.argc-1=X
```

This prints the value of `argc-1` which as a side effect sets dot; when `argc` is one the breakpoint will fire. Beware that local variables may be stored in registers; see the BUGS section.

FILES

`/proc/*/text`
`/proc/*/mem`
`/proc/*/ctl`
`/proc/*/note`

SEE ALSO

nm(1), *proc*(3)

J. F. Maranzano and S. R. Bourne, ‘A Tutorial Introduction to ADB’ in Bell Laboratories, *UNIX Programmer’s Manual,* Volume 2, Holt, Rinehart and Winston (1984)

DIAGNOSTICS

Exit status is null, unless the last command failed or returned non-null status.

BUGS

The alternate `sparc` disassembly format, `sunsparc`, reverses the order of the first two registers relative to the SUN assembler.

Examining a local variable with *routine.name* returns the contents of the memory allocated for the variable. This might return the wrong value: optimization may move the variable into a register, especially in the current stack frame. Compiling with the `-N` flag may help.

Variables and parameters that have been optimized away do not appear in the symbol table, returning the error *bad local variable* when accessed by *db.*

In some cases, the stack frame is not completely set when a breakpoint or single step stops a process in the first couple of instructions of a function. As a result, the *$c* and *$C* produce inaccurate stack traces. Stepping a couple of instructions into the function sets the stack frame and produces accurate traces.

NAME

dc − desk calculator

SYNOPSIS

`dc` [*file*]

DESCRIPTION

Dc is an arbitrary precision desk calculator. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number	The value of the number is pushed on the stack. A number is an unbroken string of the digits `0-9A-F` or `0-9a-f.` A hexadecimal number beginning with a lower case letter must be preceded by a zero to distinguish it from the command associated with the letter. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.
`+ - / * % ^`	Add `+`, subtract `-`, multiply `*`, divide `/`, remainder `%`, or exponentiate `^` the top two values on the stack. The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.
`s`*x* `S`*x*	Pop the top of the stack and store into a register named *x,* where *x* may be any character. Under operation `S` register *x* is treated as a stack and the value is pushed on it.
`l`*x* `L`*x*	Push the value in register *x* onto the stack. The register *x* is not altered. All registers start with zero value. Under operation `L` register *x* is treated as a stack and its top value is popped onto the main stack.
`d`	Duplicate the top value on the stack.
`p`	Print the top value on the stack. The top value remains unchanged. `P` interprets the top of the stack as an ASCII string, removes it, and prints it.
`f`	Print the values on the stack.
`q` `Q`	Exit the program. If executing a string, the recursion level is popped by two. Under operation `Q` the top value on the stack is popped and the string execution level is popped by that value.
`x`	Treat the top element of the stack as a character string and execute it as a string of *dc* commands.
`X`	Replace the number on the top of the stack with its scale factor.
`[ ... ]`	Put the bracketed ASCII string on the top of the stack.
<*x* >*x* =*x*	Pop and compare the top two elements of the stack. Register *x* is executed if they obey the stated relation.
`v`	Replace the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.
`!`	Interpret the rest of the line as a shell command.
`c`	Clear the stack.
`i`	The top value on the stack is popped and used as the number base for further input.
`I`	Push the input base on the top of the stack.

`o`	The top value on the stack is popped and used as the number base for further output. In bases larger than 10, each 'digit' prints as a group of decimal digits.
`O`	Push the output base on the top of the stack.
`k`	Pop the top of the stack, and use that value as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.
`z`	Push the stack level onto the stack.
`Z`	Replace the number on the top of the stack with its length.
`?`	A line of input is taken from the input source (usually the terminal) and executed.
`; :`	Used by *bc* for array operations.

The scale factor set by `k` determines how many digits are kept to the right of the decimal point. If *s* is the current scale factor, *sa* is the scale of the first operand, *sb* is the scale of the second, and *b* is the (integer) second operand, results are truncated to the following scales.

`+,-`	max(sa,sb)
`*`	min($sa+sb$, max(s,sa,sb))
`/`	s
`%`	so that dividend = divisor*quotient + remainder; remainder has sign of dividend
`^`	min($sa\times\|b\|$, max(s,sa))
`v`	max(s,sa)

EXAMPLES

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

Print the first ten values of $n!$

SEE ALSO

bc(1), *hoc*(1)

DIAGNOSTICS

x `is unimplemented`, where *x* is an octal number: an internal error.
'Out of headers' for too many numbers being kept around.
'Nesting depth' for too many levels of nested execution.

BUGS

When the input base exceeds 16, there is no notation for digits greater than `F`.

NAME

dd − convert and copy a file

SYNOPSIS

`dd` [−*option value*] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O. The options are

`ibs` *n*	Set input block size to *n* bytes (default 512).
`obs` *n*	Set output block size (default 512).
`bs` *n*	Set both input and output block size, superseding *ibs* and *obs*. If no conversion is specified, preserve the input block size instead of packing short blocks into the output buffer. This is particularly efficient since no in-core copy need be done.
`cbs` *n*	Set conversion buffer size.
`skip` *n*	Skip *n* input records before copying.
`iseek` *n*	Seek *n* records forward on input file before copying.
`files` *n*	Catenate *n* input files (useful only for magnetic tape or similar input device).
`oseek` *n*	Seek *n* records from beginning of output file before copying.
`count` *n*	Copy only *n* input records.

`conv`	`ascii`	Convert EBCDIC to ASCII.
	`ebcdic`	Convert ASCII to EBCDIC.
	`ibm`	Like `ebcdic` but with a slightly different character map.
	`block`	Convert variable length ASCII records to fixed length.
	`unblock`	Convert fixed length ASCII records to variable length.
	`lcase`	Map alphabetics to lower case.
	`ucase`	Map alphabetics to upper case.
	`swab`	Swap every pair of bytes.
	`noerror`	Do not stop processing on an error.
	`sync`	Pad every input record to *ibs* bytes.

Where sizes are specified, a number of bytes is expected. A number may end with `k` or `b` to specify multiplication by 1024 or 512 respectively; a pair of numbers may be separated by `x` to indicate a product. Multiple conversions may be specified in the style: `-conv ebcdic,ucase`.

`Cbs` is used only if `ascii`, `unblock`, `ebcdic`, `ibm`, or `block` conversion is specified. In the first two cases, *n* characters are copied into the conversion buffer, any specified character mapping is done, trailing blanks are trimmed and new-line is added before sending the line to the output. In the latter three cases, characters are read into the conversion buffer and blanks are added to make up an output record of size *n.* If `cbs` is unspecified or zero, the `ascii`, `ebcdic`, and `ibm` options convert the character set without changing the block structure of the input file; the `unblock` and `block` options become a simple file copy.

SEE ALSO

cp(1)

DIAGNOSTICS

Dd reports the number of full + partial input and output blocks handled.

NAME

deroff, delatex − remove formatting requests

SYNOPSIS

`deroff` [*option* ...] *file* ...

`delatex` *file*

DESCRIPTION

Deroff reads each file in sequence and removes all *nroff* and *troff*(1) requests and non-text arguments, backslash constructions, and constructs of preprocessors such as *eqn, pic,* and *tbl*(1). Remaining text is written on the standard output. *Deroff* follows files included by `.so` and `.nx` commands; if a file has already been included, a `.so` for that file is ignored and a `.nx` terminates execution. If no input file is given, *deroff* reads from standard input.

The options are

`-w`	Output a word list, one 'word' (string of letters, digits, and properly embedded ampersands and apostrophes, beginning with a letter) per line. Other characters are skipped. Otherwise, the output follows the original, with the deletions mentioned above.
`-i`	Ignore `.so` and `.nx` requests.
`-ms` `-mm`	Remove titles, attachments, etc., as well as ordinary *troff* constructs, from *ms*(6) or *mm* documents.
`-ml`	Same as `-mm`, but remove lists as well.

Delatex does for *tex* and *latex* (see *tex*(1)) files what `deroff -w` does for *troff* files.

SEE ALSO

troff(1), *tex*(1), *spell*(1)

BUGS

These filters are not complete interpreters of *troff* or *tex.* For example, macro definitions containing `\$` cause chaos in *deroff* when the popular `$$` delimiters for *eqn* are in effect.
Text inside macros is emitted at place of definition, not place of call.

NAME

diff − differential file comparator

SYNOPSIS

`diff` [`−efbwr`] file1 ... file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If one file is a directory, then a file in that directory with basename the same as that of the other file is used. If both files are directories, similarly named files in the two directories are compared by the method of *diff* for text files and *cmp*(1) otherwise. If more than two file names are given, then each argument is compared to the last argument as above. The `−r` option causes *diff* to recursively process similarly named subdirectories. The normal output contains lines of these forms:

n1 `a` *n3,n4*
n1,n2 `d` *n3*
n1,n2 `c` *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed,* identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

The `−b` option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal. The `−w` option causes all white-space to be removed from input lines before applying the difference algorithm.

The `−e` option produces a script of *a, c* and *d* commands for the editor *ed,* which will recreate *file2* from *file1*. The `−f` option produces a similar script, not useful with *ed,* in the opposite order. It may, however, be useful as input to a stream-oriented post-processor.

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

FILES

/tmp/diff[12]

SEE ALSO

cmp(1), *ed*(1)

DIAGNOSTICS

Exit status is `empty` for no differences, `some` for some, and `error` for trouble.

BUGS

Editing scripts produced under the `−e` or `−f` option are naive about creating lines consisting of a single '**.**'.
When running *diff* on directories, the notion of what is a text file is open to debate.

NAME

doctype − intuit command line for formatting a document

SYNOPSIS

`doctype` [*option* ...] [*file*] ...

DESCRIPTION

Doctype examines a *troff*(1) input file to deduce the appropriate text formatting command and prints it on standard output. *Doctype* recognizes input for *troff*(1), related preprocessors like *eqn*(1), and the *ms*(6) and *mm* macro packages.

Option `-n` invokes *nroff* instead of *troff.* Other options are passed to *troff.*

EXAMPLES

```
eval `{doctype chapter.?} | lp -du
```

Typeset files named `chapter.0`, `chapter.1`, ...

SEE ALSO

troff(1), *eqn*(1), *tbl*(1), *pic*(1), *grap*(1), *ms*(6), *man*(6)

BUGS

In true A.I. style, its best guesses are inspired rather than accurate.

NAME

du − disk usage

SYNOPSIS

`du` [`-a`] [`-b`*size*] [*file* ...]

DESCRIPTION

Du gives the number of Kbytes allocated to data blocks of named *files* and, recursively, of files in named directories. By default, the disk block size is assumed to be 1024 bytes. Other values can be set by the `-b` option; *size* is the number of bytes, optionally suffixed `k` to specify multiplication by 1024. If *file* is missing, the current directory is used. The count for a directory includes the counts of the contained files and directories. The `-a` option prints the number of blocks for every file in a directory. Normally counts are printed only for contained directories.

NAME

echo − print arguments

SYNOPSIS

`echo` [`−n`] [*arg ...*]

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a newline on the standard output. Option `−n` suppresses the newline.

NAME

ed − text editor

SYNOPSIS

`ed` [`-`] [`-o`] [*file*]

DESCRIPTION

Ed is a venerable text editor.

If a *file* argument is given, *ed* simulates an `e` command (see below) on that file: it is read into *ed's* buffer so that it can be edited. The options are

`-` Suppress the printing of character counts by `e`, `r`, and `w` commands and of the confirming `!` by `!` commands.

`-o` (for output piping) Write all output to the standard error file except writing by `w` commands. If no *file* is given, make `/fd/1` the remembered file; see the `e` command below.

Ed operates on a 'buffer', a copy of the file it is editing; changes made in the buffer have no effect on the file until a `w` (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single character *command,* possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command may appear on a line. Certain commands allow the addition of text to the buffer. While *ed* is accepting text, it is said to be in *input mode.* In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period **.** alone at the beginning of a line.

Ed supports the *regular expression* notation described in *regexp*(6). Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often `/`) and to `\` itself.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line.* Generally, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of each command. Addresses are constructed as follows.

1. The character **.**, customarily called 'dot', addresses the current line.
2. The character `$` addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. `'x` addresses the line marked with the name *x*, which must be a lower-case letter. Lines are marked with the `k` command.
5. A regular expression enclosed in slashes (`/`) addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.
6. A regular expression enclosed in queries `?` addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.
7. An address followed by a plus sign `+` or a minus sign `−` followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.
8. An address followed by `+` (or `−`) followed by a regular expression enclosed in slashes specifies the first matching line following (or preceding) that address. The search wraps around if necessary. The `+` may be omitted, so `0/x/` addresses the *first* line in the buffer with an `x`. Enclosing the regular expression in `?` reverses the search direction.

9. If an address begins with + or − the addition or subtraction is taken with respect to the current line; e.g. `-5` is understood to mean `.-5`.

10. If an address ends with + or −, then 1 is added (resp. subtracted). As a consequence of this rule and rule 9, the address − refers to the line before the current line. Moreover, trailing + and − characters have cumulative effect, so `--` refers to the current line less 2.

11. To maintain compatibility with earlier versions of the editor, the character ^ in addresses is equivalent to −.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma `,`. They may also be separated by a semicolon `;`. In this case the current line is set to the previous address before the next address is interpreted. If no address precedes a comma or semicolon, line 1 is assumed; if no address follows, the last line of the buffer is assumed. The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default. 'Dot' means the current line.

`(.)a`
<text>
`.` Read the given text and append it after the addressed line. Dot is left on the last line input, if there were any, otherwise at the addressed line. Address `0` is legal for this command; text is placed at the beginning of the buffer.

`(.,.)b[+-]`[*pagesize*]`[pln]`

Browse. Print a 'page', normally 20 lines. The optional + (default) or − specifies whether the next or previous page is to be printed. The optional *pagesize* is the number of lines in a page. The optional `p`, `n`, or `l` causes printing in the specified format, initially `p`. Pagesize and format are remembered between `b` commands. Dot is left at the last line displayed.

`(.,.)c`
<text>
`.` Change. Delete the addressed lines, then accept input text to replace these lines. Dot is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

`(.,.)d`

Delete the addressed lines from the buffer. Dot is set to the line following the last line deleted, or to the last line of the buffer if the deleted lines had no successor.

`e` *filename*

Edit. Delete the entire contents of the buffer; then read the named file into the buffer. Dot is set to the last line of the buffer. The number of characters read is typed. The file name is remembered for possible use in later `e`, `r`, or `w` commands. If *filename* is missing, the remembered name is used.

`E` *filename*

Unconditional `e`; see '`q`' below.

`f` *filename*

Print the currently remembered file name. If *filename* is given, the currently remembered file name is first changed to *filename.*

`(1,$)g/`*regular expression*`/`*command list*

`(1,$)g/`*regular expression*`/`

`(1,$)g/`*regular expression*

Global. First mark every line which matches the given *regular expression.* Then for every such line, execute the *command list* with dot initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must end with `\`. The '`.`' terminating input mode for an `a`, `i`, `c` command may be omitted if it would be on the last line of the command list. The commands `g` and `v` are not permitted in the command list. Any character other than space or newline may be used instead of `/` to delimit the regular expression. The second and third forms mean `g/`*regular expression*`/p`.

`(.)i`

<text>

`.` Insert the given text before the addressed line. Dot is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the *a* command only in the placement of the text.

`(.,.+1)j`

Join the addressed lines into a single line; intermediate newlines are deleted. Dot is left at the resulting line.

`(.)k`*x* Mark the addressed line with name *x,* which must be a lower-case letter. The address form `'`*x* then addresses this line.

`(.,.)l`

List. Print the addressed lines in an unambiguous way: a tab is printed as `\t`, a backspace as `\b`, backslashes as `\\`, and non-printing characters as a backslash, an `x`, and two hexadecimal digits. Long lines are folded, with the second and subsequent sub-lines indented one tab stop. If the last character in the line is a blank, it is followed by `\n`. An `l` may be appended, like `p`, to any non-I/O command.

`(.,.)m`*a*

Move. Reposition the addressed lines after the line addressed by *a*. Dot is left at the last moved line.

`(.,.)n`

Number. Perform `p`, prefixing each line with its line number and a tab. An `n` may be appended, like `p`, to any non-I/O command.

`(.,.)p`

Print the addressed lines. Dot is left at the last line printed. A `p` appended to any non-I/O command causes the then current line to be printed after the command is executed.

`(.,.)P`

This command is a synonym for `p`.

`q` Quit the editor. No automatic write of a file is done. A `q` or `e` command is considered to be in error if the buffer has been modified since the last `w`, `q`, or `e` command.

`Q` Quit unconditionally.

`($)r` *filename*

Read in the given file after the addressed line. If no *filename* is given, the remembered file name is used. The file name is remembered if there were no remembered file name already. If the read is successful, the number of characters read is printed. Dot is left at the last line read from the file.

`(.,.)s`*n*`/`*regular expression*`/`*replacement*`/`

`(.,.)s`*n*`/`*regular expression*`/`*replacement*`/g`

`(.,.)s`*n*`/`*regular expression*`/`*replacement*

Substitute. Search each addressed line for an occurrence of the specified regular expression. On each line in which *n* matches are found (*n* defaults to 1 if missing), the *n*th matched string is replaced by the replacement specified. If the global replacement indicator `g` appears after the command, all subsequent matches on the line are also replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or newline may be used instead of `/` to delimit the regular expression and the replacement. Dot is left at the last line substituted. The third form means `s`*n*`/`*regular expression*`/`*replacement*`/p`. The second `/` may be omitted if the replacement is empty.

An ampersand `&` appearing in the replacement is replaced by the string matching the regular expression. The characters `\`*n,* where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression enclosed between `(` and `)`. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of `(` starting from the left.

A literal `&`, `/`, `\` or newline may be included in a replacement by prefixing it with `\`.

`(.,.)t`*a*

Transfer. Copy the addressed lines after the line addressed by *a.* Dot is left at the last line of the copy.

`(.,.)u`

Undo. Restore the preceding contents of the current line, which must be the last line in which a substitution was made.

`(1,$)v/`*regular expression*`/`*command list*
`(1,$)v/`*regular expression*`/`
`(1,$)v/`*regular expression*

This command is the same as the global command `g` except that the command list is executed with dot initially set to every line *except* those matching the regular expression.

`(1,$)w`*filename*

Write the addressed lines to the given file. If the file does not exist, it is created with mode 666 (readable and writable by everyone). If no *filename* is given, the remembered file name, if any, is used. The file name is remembered if there were no remembered file name already. Dot is unchanged. If the write is successful, the number of characters written is printed.

`(1,$)W`*filename*

Perform `w`, but append to, instead of overwriting, any existing file contents.

`($)=` Print the line number of the addressed line. Dot is unchanged.

`!`*shell command*

Send the remainder of the line after the `!` to *rc*(1) to be interpreted as a command. Dot is unchanged.

`(.+1)`<newline>

An address without a command is taken as a `p` command. A terminal `/` may be omitted from the address. A blank line alone is equivalent to `.+1p`; it is useful for stepping through text.

If an interrupt signal (DEL) is sent, *ed* prints a `?` and returns to its command level.

When reading a file, *ed* discards NUL characters and all characters after the last newline.

FILES

`/tmp/e*`
`ed.hup` work is saved here if terminal hangs up

SEE ALSO

sam(1), *sed*(1), *regexp*(6)

DIAGNOSTICS

`?`*name* for inaccessible file; `?TMP` for temporary file overflow; `?` for errors in commands or other overflows.

NAME

emacs − editor macros

SYNOPSIS

`emacs` [*options*]

DESCRIPTION

This page intentionally left blank.

SEE ALSO

sam(1), *vi*(1)

BUGS

Many.

NAME

eqn − typeset mathematics

SYNOPSIS

`eqn` [*option ...*] [*file ...*]

DESCRIPTION

Eqn is a *troff*(1) preprocessor for typesetting mathematics on a typesetter. Usage is almost always

```
eqn file ... | troff
```

If no files are specified, these programs read from the standard input. *Eqn* prepares output for the typesetter named in the `-T`*dest* option (Postscript default, see *troff*(1)). When run with other preprocessor filters, *eqn* usually comes last.

A line beginning with `.EQ` marks the start of an equation; the end of an equation is marked by a line beginning with `.EN`. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as ‘delimiters’; text between delimiters is also *eqn* input. Delimiters may be set to characters *x* and *y* with the option `-d`*xy* or (more commonly) with `delim` *xy* between `.EQ` and `.EN`. Left and right delimiters may be identical. (They are customarily taken to be `$$`). Delimiters are turned off by `delim off`. All text that is neither between delimiters nor between `.EQ` and `.EN` is passed through untouched.

Tokens within *eqn* are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like `x` could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

Subscripts and superscripts are produced with the keywords `sub` and `sup`. Thus `x sub i` makes x_i, `a sub i sup 2` produces a_i^2, and `e sup {x sup 2 + y sup 2}` gives $e^{x^2+y^2}$.

`Over` makes fractions: `a over b` yields $\frac{a}{b}$.

`Sqrt` produces square roots: `1 over sqrt {ax sup 2 +bx+c}` results in $\frac{1}{\sqrt{ax^2+bx+c}}$.

The keywords `from` and `to` introduce lower and upper limits on arbitrary things: $\lim\limits_{n\to\infty} \sum\limits_0^n x_i$ is made with `lim from {n -> inf} sum from 0 to n x sub i .`

Left and right brackets, braces, etc., of the right height are made with `left` and `right`: `left [ x sup 2 + y sup 2 over alpha right ] ~=~1` produces $\left[x^2 + \frac{y^2}{\alpha} \right] = 1$. The `right` clause is optional. Legal characters after `left` and `right` are braces, brackets, bars, `c` and `f` for ceiling and floor, and `""` for nothing at all (useful for a right-side-only bracket).

Vertical piles of things are made with `pile`, `lpile`, `cpile`, and `rpile`: `pile {a above b above c}` produces $\begin{matrix} a \\ b \\ c \end{matrix}$. There can be an arbitrary number of elements in a pile. `lpile` left-justifies, `pile` and `cpile` center, with different vertical spacing, and `rpile` right justifies.

Matrices are made with `matrix`: `matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }` produces $\begin{matrix} x_i & 1 \\ y_2 & 2 \end{matrix}$. In addition, there is `rcol` for a right-justified column.

Diacritical marks are made with `prime`, `dot`, `dotdot`, `hat`, `tilde`, `bar`, `under`, `vec`, `dyad`, and `under`: `x sub 0 sup prime = f(t) bar + g(t) under` is $x_0' = \overline{f(t)} + \underline{g(t)}$, and `x vec = y dyad` is $\vec{x} = \overleftrightarrow{y}$.

Sizes and fonts can be changed with prefix operators `size` *n*, `size ±`*n*, `fat`, `roman`, `italic`, `bold`, or `font` *n*. Size and fonts can be changed globally in a document by `gsize` *n* and `gfont` *n*, or by the command-line arguments `-s`*n* and `-f`*n*.

Normally subscripts and superscripts are reduced by 3 point sizes from the previous size; this may be changed by the command-line argument `-p`*n*.

Successive display arguments can be lined up. Place `mark` before the desired lineup point in the first equation; place `lineup` at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with `define`: `define` *thing* `%` *replacement* `%` defines a new token called *thing* which will be replaced by *replacement* whenever it appears thereafter. The `%` may be any character that does not occur in `replacement`.

Keywords like `sum` (∑), `int` (∫), `inf` (∞), and shorthands like `>=` (≥), `->` (→), and `!=` (≠) are recognized. Greek letters are spelled out in the desired case, as in `alpha` or `GAMMA`. Mathematical words like `sin`, `cos`, `log` are made Roman automatically. *Troff*(1) four-character escapes like `\(lh` (☜) can be used anywhere. Strings enclosed in double quotes " " are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff* when all else fails.

FILES

`/sys/lib/font/devpost` font descriptions for Postscript

SEE ALSO

troff(1), *tbl*(1),
B. W. Kernighan and L. L. Cherry, 'Typesetting Mathematics—User's Guide', Unix Research System Programmer's Manual, Volume 2
J. F. Ossanna and B. W. Kernighan, 'NROFF/TROFF User's Manual', *ibid.*

BUGS

To embolden digits, parens, etc., it is necessary to quote them, as in `bold "12.3"`.

NAME

factor, primes – factor a number, generate large primes

SYNOPSIS

`factor` [*number*]

`qfactor`

`primes` [*start* [*finish*]]

DESCRIPTION

Factor prints *number* and its prime factors, each repeated the proper number of times. The number must be positive and less than 2^{56} (about 7.2×10^{16}).

If no *number* is given, *factor* reads a stream of numbers from the standard input and factors them. It exits on any input not a positive integer. Maximum running time is proportional to $\sqrt{n}$.

Primes prints the prime numbers ranging from *start* to *finish,* where *start* and *finish* are positive numbers less than 2^{56}. If *finish* is missing, *primes* prints without end; if *start* is missing, it reads the starting number from the standard input.

NAME

file − determine file type

SYNOPSIS

`file` [*file* ...]

DESCRIPTION

File performs a series of tests on its argument *files* in an attempt to classify their contents by language or purpose. If no arguments are given, the classification is performed on standard input.

BUGS

It can make mistakes, for example classifying a file of decimal data, `.01`, `.02`, etc. as *troff*(1) input.

NAME

fmt – ultra-simple text formatter

SYNOPSIS

`fmt` [*option* ...] [*file* ...]

DESCRIPTION

Fmt copies the given *files* (standard input by default) to its standard output, filling and indenting lines. The options are

`-l` *n*	Output line length is *n,* including indent, (default 70).
`-i` *n*	Indent *n* spaces (default 0).
`-j`	Don't join input lines, simply split them where necessary.

Empty lines and initial white space in input lines are preserved. Empty lines are inserted between input files.

Fmt is idempotent: it leaves already formatted text unchanged.

BUGS

Words longer than 256 characters are split.

NAME

fone − control ISDN telephone

SYNOPSIS

`fone` [*options...*]

DESCRIPTION

Fone manages an AT&T 7506 ISDN telephone set. Program control is supplementary: normal functioning of the phone is not affected. The set must be equipped with the ProPhone1.5™ ROM or equivalent. To initialize after the ROM is first installed, push the `Select`, then the `Data` button, and set parameters as follows:

```
DATA MODE: B2
DATA RATE: 19200
PARITY: SPACE
LOCAL MODE: AT
```

Fone should be run once per terminal session; it accepts commands typed in its window. If the file `call.log` exists and is writable, it will contain a log of calls. A permanent log file may be initialized with:

```
chmod +a call.log >call.log
```

Options for *fone* are:

`-f` *file*	The telephone is controlled through *file* instead of `/dev/eia0`.
`-l` *file*	Calls are logged in *file* instead of `call.log`.

Commands to *fone* are read, one per line, from the standard input. (`.,.`).TP 9 `c` *string* Call telephone number *string*. Non-alphanumeric characters are discarded, and the appropriate prefix (9, 91, or none) is guessed from the length of the result. (If there is no active call, and *string* begins with a digit, the initial `c` may be omitted.)

`C` *string*	The alphanumeric characters in *string* are dialed exactly as given.
`d`	Drop the active call, or the last party added to a conference call.
`h`	Put the active call on hold.
`k` *string*	Add a party to a conference call. The active call is placed on hold, and *string* is parsed and dialed. Once the second call is connected, the `k` command with no argument adds the new call to the previously active call appearance. If the called party is indisposed, the commands `d` and `r` will drop the second call and reconnect the first.
`q` *string*	Query the local switch for directory entries matching *string* (a surname preceded by up to two initials, like `fraser`, `a fraser`, `ag fraser`, or `a g fraser`).
`r` *id*	Reconnect a call on hold and make it the active call. If the appearance *id* is omitted, it defaults to that of the lowest numbered call on hold.
`s`	Show the *id*, *state*, and *calling information* for each call.
`t` *string*	Run the *tel*(1) command on *string*.
`x` *string*	Transfer the active call. The active call is placed in limbo (similar to hold), and *string* is parsed and dialed. Once the second call is connected, the `x` command with no argument bridges the two calls together and drops the intermediary (you). If the called party is indisposed, the commands `d` and `r` will drop the second call and reconnect the first.
`.` *string*	Transmits the converted touchtone *string* to the called party; a *string* that begins with a `*`, `#`, or a digit doesn't need the preceding `.`. (See command `c` above if no call is active.)
`?`	Print a summary of commands.

The `c` command may be given before or after lifting the handset; if the handset is down, the call is placed with the speaker on and mike muted, so you can hear what's happening.

FILES

`call.log`	log of calls
`/dev/eia0`	RS232 line to phone

BUGS

The speakerphone is not really supported.
There's still no way to set the clock.

NAME

fortune − sample lines from a file

SYNOPSIS

`fortune` [*file*]

DESCRIPTION

Fortune prints a one-line aphorism chosen at random. If a *file* is specified, the saying is taken from that file; otherwise it is selected from `/sys/games/lib/fortunes`.

FILES

`/sys/games/lib/fortunes`	
`/sys/games/lib/fortunes.index`	fast lookup table, maintained automatically

NAME

freq − print histogram of character frequencies

SYNOPSIS

`freq` [`-dxocr`] [*file ...*]

DESCRIPTION

Freq reads the given files (default standard input) and prints histograms of the character frequencies. By default, *freq* counts each byte as a character; under the `-r` option it instead counts UTF sequences, that is, runes.

Each non-zero entry of the table is printed preceded by the byte value, in decimal, octal, hex, and Unicode character (if printable). If any options are given, the `-d`, `-x`, `-o`, `-c` flags specify a subset of value formats: decimal, hex, octal, and character, respectively.

SEE ALSO

utf(6)

NAME

4s, 5s, ana, gnuchess, juggle, mandel, plumb, quiz, smiley, life, fsim, clock, catclock, fireworks, swar, zork − time wasters

SYNOPSIS

`games/4s`
`games/5s`
`games/ana` [*fixwords*]
`games/fireworks`
`games/gnuchess`
`games/juggle` [`-h` *nhand*] [*start*] *pattern*
`games/mandel`
`games/plumb` [*level*]
`games/quiz` [*category1 category2*]
`games/smiley`
`games/life` *startfile*
`games/fsim`
`games/clock`
`games/catclock` [`-c`]
`games/swar`

DESCRIPTION

There are a few games in `/bin/games`:

`4s, 5s`	Try to fill complete rows using 4-square or 5-square tiles. Move tiles left or right by moving the mouse. Rotate tiles with buttons 1 and 3. Drop tiles for more points with button 2 or the space bar.
`ana`	Find anagrams for words typed on standard input. Anagrams can contain several dictionary words. The *fixwords* argument or numbers typed on standard input fix the number of words in the output anagrams.
`gnuchess`	Play chess. Type `help` to list commands.
`juggle`	Simulate a juggler. The *pattern* lists the heights of a repeating sequence of throws. At time t, hand (t mod nhand) throws. At that time, it must hold exactly one ball, unless it executes a 0 throw, in which case it must hold no ball. A throw of height h at time t lands at time t+h in hand ((t+h) mod nhand). Some *pattern*s require a *start* sequence of throws to get the balls into position.
`mandel`	Compute and display Mandelbrot sets. Menus on the mouse buttons control various things.
`plumb`	Build a plumbing system. Keep ahead of the advancing oil and don't waste pipe. The *level* argument lets you start at a harder level.
`quiz`	gives associative knowledge tests on various subjects.
`smiley`	A game of historical importance. Type space to shoot, comma to move left, and period to go right.
`life`	Play the game of life, given an initial position. There is a library of interesting initial positions; the library is consulted if *startfile* cannot be found.
`fsim`	Pretend you're flying a Cessna.
`clock`	
`catclock`	Display analog clocks. Option `-c` makes *catclock* crosseyed.
`fireworks`	Hoist the fiery blue peter.

`swar`	Space war for two players called MCI and SPRINT. One player types `a` or `d` to turn left or right, `s` to shoot, `x` to disappear briefly (‘‘enter hyperspace’’) and `w` to accelerate. The other player uses `k`, `;`, `l`, `.` (the period) and `o`. AT&T scores whenever either ship shoots itself or otherwise causes mayhem. Hyperspace is occasionally fatal.
`zork`	The venerable adventure game.

FILES

`/sys/games/lib/4scores`	scores of *4s* games
`/sys/games/lib/5scores`	scores of *5s* games
`/sys/games/lib/plumb/scores`	scores of *plumb* games
`/sys/games/lib/anawords`	used by *ana*
`/sys/games/lib/plumb/*`	miscellaneous files used by *plumb*
`/sys/games/lib/quiz/*`	miscellaneous files used by *quiz*
`/sys/games/lib/life/*`	interesting starting positions

NAME

gcan, homespool, gspool − interface to gnot laser-printers

SYNOPSIS

`gcan` [*option* ...] [*file* ...]

`homespool`

`gspool`

DESCRIPTION

This command prints *files* (standard input by default) on a gnot laser printer. *Troff*(1) output and ordinary text are generally recognized and processed appropriately; in any case the first option can be used to specify the type of input:

`−sb`	Bitmap files. (See *bitmap*(6)).
`−sc`	ordinary text.
`−sd`	*troff* output.
`−sm`	MIDI output.
`−sr`	`TYPE=rle` run-length-encoded bitmaps.

The destination printer is determined in one of the following ways, in decreasing order of precedence:

option `−d` *dest*
environment variable `candest`
printer named in file `/sys/lib/candest`

These options are common to all classes of input:

`--`	No more options.
`-dg`	Destination is printer outside of graphics lab.
`-du`	Destination is printer in dirty room.
`-dt`	Destination is printer in 2C465 (tom's office).
`-p`	Output goes to `/usr/$user/pspool` for proofreading.
`-f`*file*	Output goes to *file* for debugging.
`-G`	Output goes to `/usr/$user/spool` for local printing.
`-H`	Suppress header page.
`-I`	Do not attempt page reversal (no page number index is generated).
`-l`*n*	Set number of lines per page (forces `c` service even if data look like *troff*).
`-%`*n*	Print odd-numbered pages if *n*=1, even-numbered pages if *n*=2.
`-D`	Turn on debugging.
`-o...`	Print only those pages whose numbers appear in the list (e.g., `−o 1,3,5−8`).
`-O...`	Like `−o`, except pages are counted sequentially, i.e., *troff*-assigned page numbers are ignored. (Both options have the same effect on non-*troff* files.)

These options are only valid with the types of input indicated:

`−m`*n*	Set magnification to *n* (type `b`).
`−s`*n*	Set point size to *n* (type `c`).
`−x`*n*	Move the image *n* pixels left (types `c`, `d`, `r`).
`−y`*n*	Move the image *n* pixels down (types `c`, `d`, `r`).

`homespool` initializes the files needed for a private spooler; `gspool` starts such a spooler. See the `−G` option.

FILES

`/sys/lib/candest`	name of default printer.
`/usr/$user/pspool`	proofreading spool area.

`/usr/$user/spool`	spool area.

SEE ALSO

troff(1).

BUGS

More input classes should be divined from the data.

NAME

grap − pic preprocessor for drawing graphs

SYNOPSIS

`grap` [*file* ...]

DESCRIPTION

Grap is a *pic*(1) preprocessor for drawing graphs on a typesetter. Graphs are surrounded by the *troff* 'commands' `.G1` and `.G2`. Data are scaled and plotted, with tick marks supplied automatically. Commands exist to modify the frame, add labels, override the default ticks, change the plotting style, define coordinate ranges and transformations, and include data from files. In addition, *grap* provides the same loops, conditionals, and macro processing that *pic* does.

`frame ht` *e* `wid` *e* `top dotted` ...: Set the frame around the graph to specified `ht` and `wid`; default is 2 by 3 (inches). The line *styles* (`dotted`, `dashed`, `invis`, `solid` (default)) of the *sides* (`top`, `bot`, `left`, `right`) of the frame can be set independently.

`label` *side* `"a label" "as a set of strings"` *adjust*: Place label on specified side; default side is bottom. *adjust* is `up` (or `down left right`) *expr* to shift default position; `width` *expr* sets the width explicitly.

`ticks` *side* `in at` *optname expr, expr, ...*: Put ticks on *side* at *expr, ...,* and label with *"expr".* If any *expr* is followed by "...", label tick with "...", and turn off all automatic labels. If "..." contains `%f`'s, they will be interpreted as `printf` formatting instructions for the tick value. Ticks point `in` or `out` (default out). Tick iterator: instead of `at` ..., use `from` *expr* `to` *expr* `by` *op expr* where *op* is optionally `+-*/` for additive or multiplicative steps. `by` can be omitted, to give steps of size 1. If no ticks are requested, they are supplied automatically; suppress this with `ticks off`. Automatic ticks normally leave a margin of 7% on each side; set this to anything by `margin` = *expr.*

`grid` *side linedesc* `at` *optname expr, expr, ...*: Draw grids perpendicular to *side* in style *linedesc* at *expr,* .... Iterators and labels work as with ticks.

`coord` *optname* `x` *min, max* `y` *min, max* `log x log y`: Set range of coords and optional log scaling on either or both. This overrides computation of data range. Default value of *optname* is current coordinate system (each `coord` defines a new coordinate system).

`plot` *"str"* `at` *point*; *"str"* `at` *point*: Put *str* at *point.* Text position can be qualified with `rjust`, `ljust`, `above`, `below` after "...".

`line from` *point* `to` *point linedesc*: Draw line from here to there. `arrow` works in place of `line`.

`next` *optname* `at` *point linedesc*: Continue plot of data in *optname to point*; default is current.

`draw` *optname linedesc ...*: Set mode for `next`: use this style from now on, and plot "..." at each point (if given).

`new` *optname linedesc ...*: Set mode for `next`, but disconnect from previous.

A list of numbers *x y1 y2 y3 ...* is treated as `plot bullet at` *x,y1*; `plot bullet at` *x,y2*; etc., or as `next at` *x,y1* etc., if `draw` is specified. Abscissae of 1,2,3,... are provided if there is only one input number per line.

A point *optname expr, expr* maps the point to the named coordinate system. A *linedesc* is one of `dot dash invis solid` optionally followed by an expression.

`define` *name* `{` *whatever* `}`: Define a macro. There are macros already defined for standard plotting symbols like `bullet`, `circle`, `star`, `plus`, etc., in `/sys/lib/grap.defines`, which is included if it exists.

var = *expr*: Evaluate an expression. Operators are `+ - *` and `/`. Functions are `log` and `exp` (both base 10), `sin`, `cos`, `sqrt`; `rand` returns random number on [0,1); `max(`*e*`,`*e*`)`, `min(`*e*`,`*e*`)`, `int(`*e*`)`.

`print` *expr*; `print "..."`: As a debugging aid, print *expr*or *string* on the standard error.

`copy` `"`*file name*`"`: Include this file right here.

`copy thru` *macro*: Pass rest of input (until `.G2`) through *macro*, treating each field (non-blank, or "...") as an argument. *macro* can be the name of a macro previously defined, or the body of one in place, like `/plot $1 at $2,$3/`.

`copy thru` *macro* `until` `"`*string*`"`: Stop copy when input is *string* (left-justified).

`pic` *remainder of line*: Copy to output with leading blanks removed.

`graph` *Name pic-position*: Start a new frame, place it at specified position, e.g., `graph Thing2 with .sw at Thing1.se + (0.1,0)`. *Name* must be capitalized to keep *pic* happy.

`.` *anything at beginning of line*: Copied verbatim.

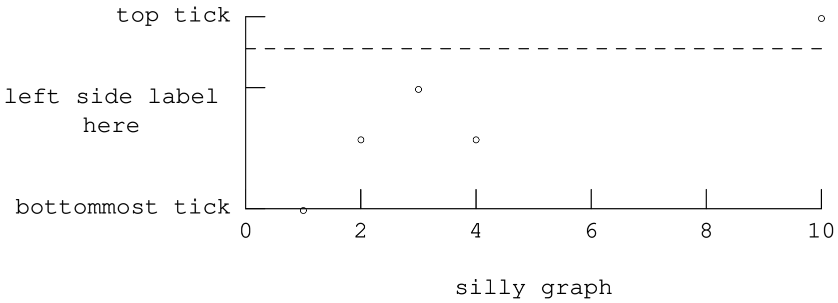
`sh %`*anything* `%`: Pass everything between the `%`'s to the shell; as with macros, `%` may be any character and *anything* may include newlines.

`#` *anything*: A comment, which is discarded.

Order is mostly irrelevant; no category is mandatory. Any arguments on the `.G1` line are placed on the generated `.PS` line for *pic.*

EXAMPLES

```
.G1
frame ht 1 top invis right invis
coord x 0, 10 y 1, 3 log y
ticks left in at 1 "bottommost tick", 2,3 "top tick"
ticks bot in from 0 to 10 by 2
label bot "silly graph"
label left "left side label" "here"
grid left dashed at 2.5
copy thru / circle at $1,$2 /
1 1
2 1.5
3 2
4 1.5
10 3
.G2
(.,.)... 0.000i 1.583i 4.700i 0.000i
```



FILES

`/sys/lib/grap.defines`	definitions of standard plotting characters, e.g., bullet

SEE ALSO

pic(1), *troff*(1)

J. L. Bentley and B. W. Kernighan, 'GRAP—A Language for Typesetting Graphs', Unix Research System Programmer's Manual, Volume 2

NAME

graph − draw a graph

SYNOPSIS

`graph` [*option* ...]

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas (*x*-values) and ordinates (*y*-values) of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by *plot*(1) filters.

If an ordinate is followed by a nonnumeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes `" "` in which case they may be empty or contain blanks and numbers; labels never contain newlines.

The following options are recognized, each as a separate argument.

`-a`	Supply abscissas automatically; no *x*-values appear in the input. Spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0, or 1 with a log scale in *x,* or the lower limit given by `-x`).
`-b`	Break (disconnect) the graph after each label in the input.
`-c`	Character string given by next argument is default label for each point.
`-g`	Next argument is grid style, 0 no grid, 1 frame with ticks, 2 full grid (default).
`-l`	Next argument is a legend to title the graph. Grid ranges are automatically printed as part of the title unless a `-s` option is present.
`-m`	Next argument is mode (style) of connecting lines: 0 disconnected, 1 connected. Some devices give distinguishable line styles for other small integers. Mode −1 (default) begins with style 1 and rotates styles for successive curves under option `-o`.
`-o`	(Overlay.) The ordinates for *n* superposed curves appear in the input with each abscissa value. The next argument is *n*.
`-s`	Save screen; no new page for this graph.
`-x l`	If `l` is present, *x*-axis is logarithmic. Next 1 (or 2) arguments are lower (and upper) *x* limits. Third argument, if present, is grid spacing on *x* axis. Normally these quantities are determined automatically.
`-y l`	Similarly for *y*.
`-e`	Make automatically determined *x* and *y* scales equal.
`-h`	Next argument is fraction of space for height.
`-w`	Similarly for width.
`-r`	Next argument is fraction of space to move right before plotting.
`-u`	Similarly to move up before plotting.
`-t`	Transpose horizontal and vertical axes. (Option `-a` now applies to the vertical axis.)

If a specified lower limit exceeds the upper limit, the axis is reversed.

SEE ALSO

plot(1), *grap*(1)

BUGS

Segments that run out of bounds are dropped, not windowed.
Logarithmic axes may not be reversed.
Option `-e` actually makes automatic limits, rather than automatic scaling, equal.

NAME

grep − search a file for a pattern

SYNOPSIS

`grep` [*option ...*] *pattern* [*file ...*]

DESCRIPTION

Grep searches the input *files* (standard input default) for lines (with newlines excluded) that match the *pattern,* a regular expression as defined in *regexp*(6). Normally, each line matching the pattern is 'selected', and each selected line is copied to the standard output. The options are

`-c`	Print only a count of matching lines.
`-h`	Do not print file name tags (headers) with output lines.
`-i`	Ignore alphabetic case distinctions. The implementation folds into lower case all letters in the pattern and input before interpretation. Matched lines are printed in their original form.
`-l`	(ell) Print the names of files with selected lines; don't print the lines.
`-L`	Print the names of files with no selected lines; the converse of `-l`.
`-n`	Mark each printed line with its line number counted in its file.
`-s`	Produce no output, but return status.
`-v`	Reverse: print lines that do not match the pattern.

Output lines are tagged by file name when there is more than one input file. (To force this tagging, include `/dev/null` as a file name argument.)

Care should be taken when using the shell metacharacters `$*[^|()=\` and newline in *pattern*; it is safest to enclose the entire expression in single quotes `'...'`.

SEE ALSO

ed(1), *awk*(1), *sed*(1), *sam*(1), *regexp*(6)

DIAGNOSTICS

Exit status is null if any lines are selected, or non-null when none are selected or an error occurs.

NAME

help − experimental window system

SYNOPSIS

`help/help`

`help/buf`

`` eval `{/bin/help/parse [ -ca0 ] } ``

DESCRIPTION

Help is an experimental combination of window system, editor, shell, and user interface. *Help* supports textual applications only.

Layout

The screen is divided initially into two columns of windows. Each window is divided into two components: a single line of text across the top, called the *tag*, and a multi-line *body* of text below. Typically a window represents a file whose name is in the tag and whose contents are in the body. In this case, the file name appears as the first part of the tag, followed by white space. The *directory name* associated with the window is the file name in the tag stripped of text after its final slash character.

The columns also have tags. The windows and columns are stacked like aligned sheets of paper that may overlap. (In the rest of this document, 'window' will stand for 'window or column'.) Each window has associated a small black square to the left of the column containing the window; each column has one to the top. Clicking with mouse button 1 on the square brings the associated window to the top of the stack of windows in which it resides, leaving its x-y position unchanged.

Pressing mouse button 3 in the tag of a window allows the window to be moved: hold the button down and release it in the desired new position.

New windows are placed automatically by *help*. They are placed in the bottom of the column containing the selected text (q.v.).

Text

Each window contains editable text in its tag and body. The behavior of this text is essentially the same as in *sam*(1): button 1 selects text; typing replaces the selection; there is a scroll bar to the left, etc. Double clicking button 1 selects as in *sam.* The tag and body have independent selections.

The current window (tag or body) is the one under the mouse; there is no 'click-to-type' property.

Cutting and pasting are done by chords on the mouse buttons. While button 1 is held down after a selection, clicking button 2 will cut the selected text and button 3 will paste the snarf buffer into the selected text. There is no menu for cutting and pasting, although there are such commands; see the next section.

Execution

Commands are executed by selecting the text of the command using button 2. When the button is released, the button 2 selection is passed to the shell for execution. While a command is executing, the command name appears in the top line of the *help* screen; when the command completes, it is removed. Selecting with button 2 does not affect the current text.

Some commands are built in to *help.* By convention, these commands have initial capital letters. Examples are `Cut`, `Paste`, `Snarf`, `Open`, etc. Such commands are interpreted internally, much like functions in *rc*(1), and do not correspond to executable files. Some built-ins take no arguments and cause actions to the window in which they are executed. These are suffixed with an exclamation point. They are `Close!`, which closes the window, `Get!`, which rereads the file, and `Put!`, which writes the window to its file.

The file name to execute is found as follows. If the command name begins `/` or `./` it is taken as a literal file name. If not, the name is prefixed by the directory name of the window holding the command name. If that file does not exist or if there is no directory name, the name is prefixed by `/bin/`.

Multiple words are interpreted as a command to execute followed by its arguments. Single quotes behave as in *rc*(1). Built-in commands that take an argument will use the selected text as an argument if no argument is provided explicitly. For example, one may select a file name with button 1 and select `Open` with button 2 to open a window on a file. Shell commands must be provided their arguments explicitly, although it is possible to discover the selection in a program designed to run under *help*; see *help*(4) and read about *parse* below.

Defaults

When selecting text with button 1, double clicks select words, lines, etc. There is no way to double click with button 2, however. Instead, a null selection — a click — with button 2 on a word expands to the entire white-space-delimited word containing the selection. For example, clicking button 2 on `Cut` will cut the selected text. If the text selected with button 2 is not a null string, no such expansion occurs.

When a null button 1 selection is used as an argument to a command, rules relevant to the command are used to expand the selection. For example, `Open` expands the selection to a white-space-delimited word and interprets that as a file name. It is therefore possible to open a file with two clicks: one on the file name, one on `Open`. Also, since typing leaves the selection at the null string at the end of the typed text, one may load a new file by typing its name (leaving the selection at the end of the name) and clicking button 2 on `Open.` Other commands interpret the selection as numbers, words, C identifiers, etc., *according to the rules of the command.*

Tools

Tools in *help* comprise directories holding executable programs and associated file, conventionally called `stf` ('stuff') that holds the templates for executing the programs. Standard tools live in `/help`. For example, `/help/cbr` is a directory of tools for browsing C source text and `/help/cbr/stf` is a text file that acts somewhat like a menu of browser commands. By the rules mentioned above, clicking button 2 on, say, `decl` in the window holding `/help/cbr/stf` will execute `/help/cbr/decl`, a program that identifies the declaration of a C variable.

Boot

When started, *help* initializes its display and prints the words `help/Boot` and `Exit` across the top. `Exit` is naturally the built-in command to quit *help*. The program `help/Boot` loads *help* with the tools named in the environment variable `$helpboot .`

Support programs

See *help*(4) for an explanation of the control files *help* offers its applications. Two programs in `/bin/help` assist such applications. *Buf* collects its input and emits it in a single `write` system call. It sends a maximum of 8192 bytes. *Parse* reads *help*'s control files to find the selected text. It expands the selection, if null, to the alphanumeric word defined by its option: `-0` (zero) selects numbers; `-a` selects alphabetic words; and `-c` selects C identifiers. *Parse* then prints strings to set the environment variables `$file` to the file name of the window holding the selection, `$dir` to the directory, `$base` to the basename (file minus directory), `$line` to the line number holding the beginning of the selection, and `$id` to the text containing the word. The output of *parse* should be executed by `rc`.

FILES

`/mnt/help`	Files served by *help* (also unioned in `/dev` in a window's name space, before the terminal's real `/dev` files).
`/help`	Directory of tools.
`/help/lib/boot`	Bootstrap program.

SEE ALSO

8½(1), *8½*(4), *help*(4)
Rob Pike, *A Global Minimal User Interface.*

BUGS

Help has not been engineered or tested nearly as well as *8½*.

NAME

hoc − interactive floating point language

SYNOPSIS

`hoc` [*file* ...]

DESCRIPTION

Hoc interprets a simple language for floating point arithmetic, at about the level of BASIC, with C-like syntax and functions.

The named *files* are read and interpreted in order. If no *file* is given or if *file* is − *hoc* interprets the standard input.

Hoc input consists of *expressions* and *statements*. Expressions are evaluated and their results printed. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call *print*.

Variable names have the usual syntax, including _; the name _ by itself contains the value of the last expression evaluated. The variables `E`, `PI`, `PHI`, `GAMMA` and `DEG` are predefined; the last is 59.25..., degrees per radian.

Expressions are formed with these C-like operators, listed by decreasing precedence.

`^` exponentiation

`! - ++ --`

`* / %`

`+ -`

`> >= < <= == !=`

`&&`

`||`

`= += -= *= /= %=`

Built in functions are `abs`, `acos`, `asin`, `atan` (one argument), `cos`, `cosh`, `erf`, `erfc`, `exp`, `gamma`, `int`, `log`, `log10`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`. The function `read(x)` reads a value into the variable `x` and returns 0 at EOF; the statement `print` prints a list of expressions that may include string constants such as `"hello\n"`.

Control flow statements are `if-else`, `while`, and `for`, with braces for grouping. Newline ends a statement. Backslash-newline is equivalent to a space.

Functions and procedures are introduced by the words `func` and `proc`; `return` is used to return with a value from a function. Within a function or procedure, arguments are referred to as `$1`, `$2`, etc.; all other variables are global.

EXAMPLES

```
func gcd() {
      temp = abs($1) % abs($2)
      if(temp == 0) return abs($2)
      return gcd($2, temp)
}
for(i=1; i<12; i++) print gcd(i,12)
```

SEE ALSO

bc(1), *dc*(1)
B. W. Kernighan and R. Pike, *The Unix Programming Environment,* Prentice-Hall, 1984

BUGS

Error recovery is imperfect within function and procedure definitions.

NAME

hp − emulate an HP 2621 terminal

SYNOPSIS

`hp`

DESCRIPTION

hp replaces an 8½ window with an emulation of an HP 2621 terminal.

BUGS

Hp cannot resize a window. If a 24x80 screen is required, it can draw an outline (using a menu item on button 2) and will use only the space within the outline, but the user is responsible for resizing the window to fit the outline.

Use care in setting echo and newline modes when connecting to Unix systems via *con.* It may also be necessary to set the emulator into raw mode manually (using a button 2 menu entry).

NAME

kana8½ − language transliterator

SYNOPSIS

`kana8½` [*args*]

DESCRIPTION

The *kana8½* script starts an instance of the window manager *8½*(1) able to transliterate keyboard letter sequences into characters in languages that do not use the Latin character set. It also creates a small control window; a menu on button 2 in this window chooses among English (default), Russian, Japanese hiragana or katakana, or Greek. Language may also be controlled by sending one of `english`, `katakana`, `hiragana`, `russian`, or `greek` to the file `/mnt/kanactl/data`.

The Japanese selections interpret lower-case letters as a Hepburn representation of hiragana or katakana. The Russian selection interprets letters as Cyrillic; the transliteration is mostly phonetic, with `'` for *myagkij-znak*, `''` for *tverdyj-znak* (spell it *tvyordyj-znak* if you want the dieresis), `j` for *i-kratkaya*. No transliteration is done in English mode.

If no arguments are given, the font specification `unicode.9` is used; if there are arguments, they are passed unchanged to the window system and an appropriate font must already be installed.

The command is a short script that starts `aux/kana`; this program inserts itself between `/dev/cons` and the window system, and performs the transliteration. The script then invokes an instance of *8½*.

FILES

`/mnt/kana/data1`	used as a naming point for binding output upon `/dev/cons`.
`/mnt/kanactl/data`	for controlling language.

SEE ALSO

8½(1)

BUGS

Considerably more sophistication is required to support ideographic languages.

The language can't be selected independently in each window.

NAME

kill, broke − print commands to kill processes

SYNOPSIS

`kill` *name*

`broke`

DESCRIPTION

Kill prints commands that will cause all processes called *name* and owned by the current user to be terminated. Use the `send` command of *8½*(1), or pipe the output of *kill* into *rc*(1) to execute the commands.

Kill suggests sending a `kill` note to the process; the same message delivered to the process's `ctl` file (see *proc*(3)) is a surer, if heavy handed, kill, but is necessary if the offending process is ignoring notes.

Broke prints commands that will cause all processes in the *Broken* state and owned by the current user to go away. When a process dies because of an error caught by the system, it may linger in the *Broken* state allowing examination with a debugger. Executing the commands printed by *broke* lets the system reclaim the resources used by the broken processes.

SEE ALSO

ps(1), *stop*(1), *proc*(3)

NAME

lex − generator of lexical analysis programs

SYNOPSIS

`lex` [`-tvn`] [*file* ...]

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text. The input *files* (standard input default) contain regular expressions to be searched for and actions written in C to be executed when expressions are found.

A C source program, `lex.yy.c` is generated. This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The options have the following meanings.

`-t`	Place the result on the standard output instead of in file `lex.yy.c`.
`-v`	Print a one-line summary of statistics of the generated analyzer.
`-n`	Opposite of `-v`; `-n` is default.

EXAMPLES

This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A-Z] putchar(yytext[0]+'a'-'A');
[ ]+$
[ ]+  putchar(' ');
```

FILES

`lex.yy.c`	output
`/sys/lib/lex/ncform`	template

SEE ALSO

yacc(1), *sed*(1)

M. E. Lesk and E. Schmidt, ‘LEX—Lexical Analyzer Generator’, Unix Research System Programmer’s Manual, Volume 2

BUGS

Cannot handle UTF.

The asteroid to kill this dinosaur is still in orbit.

NAME

look − find lines in a sorted list

SYNOPSIS

`look` [`-dfnixt`*c*] [*string*] [*file*]

DESCRIPTION

Look consults a sorted *file* and prints all lines that begin with *string*. It uses binary search.

The following options are recognized. Options `dfnt` affect comparisons as in *sort*(1).

`-i` Interactive. There is no *string* argument; instead *look* takes lines from the standard input as strings to be looked up.

`-x` Exact. Print only lines of the file whose key matches *string* exactly.

`-d` 'Directory' order: only letters, digits, tabs and blanks participate in comparisons.

`-f` Fold. Upper case letters compare equal to lower case.

`-n` Numeric comparison with initial string of digits, optional minus sign, and optional decimal point.

`-t`*[c]* Character *c* terminates the sort key in the *file*. By default, tab terminates the key. If *c* is missing the entire line comprises the key.

If no *file* is specified, `/lib/words` is assumed, with collating sequence `df`.

FILES

`/lib/words`

SEE ALSO

sort(1), *grep*(1)

DIAGNOSTICS

The exit status is `"not found"` if no match is found, and `"no dictionary"` if *file* or the default dictionary cannot be opened.

NAME

lp − printer output

SYNOPSIS

`lp` [*option ...*] [*file ...*]

DESCRIPTION

Lp is a generalized output printing service. It can be used to queue files for printing, check a queue, or kill jobs in a queue. The options are:

`-d` *dest*	Select the destination printer. If *dest* is `?`, list the currently available printers. In the absence of `-d`, the destination is taken from the environment variable `LPDEST`. Destination `stdout` is the standard output. Destination `safari` is `/dev/lpt1data` line printer port on a 386 machine.
`-p` *proc*	The given preprocessor is invoked. The default preprocessor is `generic`, which tries to do the right thing for regular text, *troff*(1) output, or *tex*(1) output. If no preprocessing is desired `noproc` may be specified.
`-q`	Print the queue for the given destination. For some devices, include printer status.
`-k`	Kill the job(s) given as subsequent arguments instead of file names for the given destination.

The remaining options may be used to affect the output at a given device. These options may not be applicable to all devices.

`-c` *n*	Print *n* copies.
`-f` *font*	Set the font (default `CW.11`).
`-H`	Suppress printing of header page.
`-i` *n*	Select paper input tray *n*.
`-l` *n*	Set the number of lines per page to *n*.
`-L`	Print pages in landscape mode (i.e. turned 90 degrees).
`-m` *v*	Set magnification to *v*.
`-n` *n*	Print *n* logical pages per physical page.
`-o` *list*	Print only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *n*−*m* means pages *n* through *m*; a range −*n* means from the beginning to page *n*; a range *n*− means from page *n* to the end.
`-r`	Reverse the order of page printing (currently not functional).
`-x` *v*	Set the horizontal offset of the print image, measured in inches.
`-y` *v*	Set the vertical offset of the print image, measured in inches.

EXAMPLES

```
eqn paper | troff -ms | lp -dsafari
```

Typeset and print a paper containing equations.

```
pr -l100 file | lp -l100 -fCW.8
```

Print a file in a small font at 100 lines per page.

```
lp -dstdout /dev/windows/3/window > doc.ps
```

Convert a bitmap to a postscript file.

SEE ALSO

lp(8)
A Guide to the Lp Printer Spooler, P. Glick, *Unix Programmer's Manual,* Tenth Edition, Volume 2.

BUGS

Not all options work with all output devices. Any user can kill any job. *Lp* will accept jobs from BSD style lpdaemons but cannot send jobs to such systems.

NAME

ls, lc − list contents of directory

SYNOPSIS

`ls` [`-dlnpqrstu`] *name* ...

`lc` [`-dlnpqrstu`] *name* ...

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. When no argument is given, the current directory is listed. By default, the output is sorted alphabetically by name.

Lc is the same as *ls,* but sets the `-p` option and pipes the output through *mc*(1).

There are a number of options:

`-d`	If argument is a directory, list it, not its contents.
`-l`	List in long format, giving mode (see below), file system type (e.g., for devices, the `#` code letter that names it; see *Intro*(4)), the instance or subdevice number, owner, group, size in bytes, and time of last modification for each file.
`-n`	Don't sort the listing.
`-p`	Print only the final path element of each file name.
`-q`	List the *qid* (see *stat*(2)) of each file.
`-r`	Reverse the order of sort.
`-s`	Give size in Kbytes for each entry.
`-t`	Sort by time modified (latest first) instead of by name.
`-u`	Under `-t` sort by time of last access; under `-l` print time of last access.
`-F`	Add the character `/` after all directory names and the character `*` after all executable files.

The mode printed under the `-l` option contains 11 characters, interpreted as follows: the first character is

`d`	if the entry is a directory;
`a`	if the entry is an append-only file;
`-`	if the entry is a plain file.

The next letter is `l` if the file is exclusive access (one writer or reader at a time).

The last 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

`r`	if the file is readable;
`w`	if the file is writable;
`x`	if the file is executable;
`-`	if none of the above permissions is granted.

SEE ALSO

stat(2) *mc*(1)

NAME

mail, edmail, sendmail, seemail, aliasmail, smtp, smtpd, uk2uk, vwhois, vismon − mail commands

SYNOPSIS

`mail` [*arg ...*]

`upas/edmail` [`-cmpre`] [`-[fF]` *mfile*]

`upas/sendmail` [`-x#`] *person ...*

`seemail` [`-as`] [`-f` *file*]

`upas/aliasmail` *name ...*

`smtp` [`-fdu`] [`-h`*host*] [*.domain*] *address sender rcpt-list*

`smtpd` [`-d`]

`uk2uk` *system user*

`vwhois` *people ...*

`vismon` *system*

DESCRIPTION

Mail

Mail invokes *edmail* `-m` when no *persons* appear on the command line. It invokes *sendmail* otherwise.

Mailbox Editing

Edmail edits a mailbox. The default mailbox is `/mail/box/`*username*`/mbox`. The `-f` and `-F` command line options and the `s` and `S` editing commands specify an alternate mailbox. Unrooted path names are interpreted relative to `/mail/box/`*username* for `-f` and `s` and relative to the current directory for `-F` and `S`. If the *mfile* argument is omitted, the name defaults to `stored`.

The options for *edmail* are:

`-c`	Create a mailbox.
`-r`	Reverse: print mail in first-in, first-out order.
`-p`	Print all the mail messages without prompting for commands.
`-m`	Use a manual style of interface, that is, print no messages unless directed to.
`-f` *mfile*	Read messages from the specified file (see above) instead of the default mailbox.
`-F` *mfile*	same as `-f` with different starting point for relative paths (see above).
`-e`	Check silently if there is anything in the mailbox; return zero (true) if so, non-zero otherwise.

Edmail prints messages one at a time, prompting between messages. After printing a prompt *edmail* reads a line from the standard input to direct disposition of the message. Commands, as in *ed*(1), are of the form ‘[*range*] *command* [*arguments*]’. The command is applied to each message in the (optional) range addressed by message number and/or regular expressions in the style of *ed*(1). A regular expression in slashes searches among header (postmark) lines; an expression in percent signs searches on message content.

address	to indicate a single message header
address`,` *address*	to indicate a range of contiguous message headers
`g/`*expression*`/`	to indicate all message headers matching the regular *expression.*

The commands are:

`b`	Print the headers for the next ten messages.
`d`	Mark message to be deleted upon exiting *edmail.*
`h`	Print the disposition, size in characters, and header line of the message.
`m` *person ...*	Mail the message to the named *persons.*
`M` *person ...*	Same as `m` except that lines typed on the terminal (terminated by `EOT`) are prefixed to the message.

`p`	Print message. An interrupt stops the printing.
`r`	Reply to the sender of the message.
`R`	Like `r` but with the message appended to the reply.
`s` *mfile*	(Save) Append the message to the specified mailbox (see above).
`S` *mfile*	Same as `s` with different starting point for relative paths (see above).
`q`	Put undeleted mail back in the mailbox and stop.
EOT (control-D)	Same as `q`.
`w` *file*	Same as `s` with the mail header line(s) stripped.
`W` *file*	Same as `w` with different starting point for relative paths (see above).
`u`	Remove mark for deletion.
`x`	Exit, without changing the mailbox file.
`?`	Print a command summary.
| *command*	Run the *command* with the message as standard input.
! *command*	Escape to the shell to do *command.*
=	Print the number of the current message.

Sending Mail

Sendmail takes the standard input up to an end-of-file and adds it to each *person's* mailbox. When running in an *8½*(1) window, *sendmail* automatically puts the window into Hold mode (see *8½*(1)); this means that previous lines of the message can be edited freely, because nothing will be sent to *sendmail* until the ESC key is hit to exit Hold mode. With option `-#`, *sendmail* does not send mail, but instead reports what command would be used to send the mail. With option `-x`, *sendmail* does not send mail, but instead reports the full mail address of the recipient.

The message is automatically postmarked with the sender's name and date. Lines that look like postmarks are prefixed with >.

Person is a login name on the local system, a name for which there is an *alias*, or a network mail address.

Addressing Conventions

The local convention for converting addresses is given by rewrite rules in `/mail/lib/rewrite`. The conventions generally used are:

- A *person* containing no `!` or `@` is considered a local user or local alias. It is passed as an argument to *aliasmail* which returns either the expanded alias or `local!`*person* if there is no alias of that name.
- A canonical network mail address has the form *machine*!...!*name,* with one or more machines mentioned.

Aliasmail

Aliasmail expands mail aliases, its arguments, according to alias files. Each line of an alias file begins with `#` (comment) or with a name. The rest of a name line gives the expansion. The expansion may contain multiple addresses and may be continued to another line by appending a backslash. Items are separated by white space.

In expanding a name, the sender's personal alias file `/mail/box/`*username*`/names` is checked first. Then the system alias files, listed one per line in `/mail/lib/namefiles`, are checked in order. If the name is not found, the expansion is taken to be `local!`*name.*

Mailboxes

Incoming mail for a user *username* is put in the file `/mail/box/`*username*`/mbox` unless either the file `/mail/box/`*username*`/forward` or `/mail/box/`*username*`/pipeto` exists. The mailbox must have append-only and exclusive-access mode (see *chmod*(1)). A user must create his or her own mailbox using the `-c` option of *edmail*. Mailboxes are created writable (append-only) but not readable by others.

Forwarding

If the file `/mail/box/`*username*`/forward` exists and is readable by everyone, incoming mail will be forwarded to the addresses contained in the first line of the file. The file may contain multiple addresses.

Forwarding loops are caught and resolved by local delivery.

Filtering

If the file `/mail/box/`*username*`/pipeto` exists and is readable and executable by everyone, it will be run for each incoming message for the user. The message will be piped to it rather than appended to his/her mail box. The file is run as user `none`.

Misc

The *seemail* command notifies when a new message arrives in your mailbox. It reads a log *file*, default `/sys/log/mail`, of incoming messages. It runs continuously where it is invoked, displaying the names and icons of senders of new messages. The `-a` flag causes it to initialize by displaying all the faces in the log; `-s` causes it to overwrite multiple appearances of the same face rather than repeatedly displaying it. *Vwhois* just displays in the *seemail* window the icons of *people. Vismon* is a version of *seemail* that connects to a remote Unix (not Plan 9) system to look for mail arriving there.

Smtp sends the mail message from standard input to the users *rcpt-list* on the host at network address *address* using the Simple Mail Transfer Protocol. The return address of the mail will contain the local system name from the environment variable *sysname* and the user *sender*. If *.domain* is given, it is appended to the end of the system name. The `-u` option sends the mail in the standard Unix format instead of RFC822 format. The `-f` flag just prints out the converted message rather than sending it to the destination. The `-d` option turns on debugging output to standard error.

Smtpd receives a message using the Simple Mail Transfer Protocol. Standard input and output are the protocol connection. The `-d` option turns on debugging output to standard error. *Smtpd* is normally run by a network listener such as *listen*(8).

`uk2uk` is used by the mail rewrite rules to turn a JANET style name into a domain style name, by flipping all the components of *system,* appending !*user* to it, and writing the result to standard output.

FILES

`/sys/log/mail`	mail log file
`/mail/box/*`	mail directories
`/mail/box/*/mbox`	mailbox files
`/mail/box/*/forward`	forwarding address(es)
`/mail/box/*/pipeto`	mail filter
`/mail/box/*/L.reading`	mutual exclusion lock for multiple mbox readers
`/mail/box/*/L.mbox`	mutual exclusion lock for altering mbox
`/mail/box/*/dead.letter`	unmailable text
`/mail/box/*/names`	personal alias files
`/mail/lib/rewrite`	rules for handling addresses
`/mail/lib/namefiles`	lists files to search for aliases in
`/lib/face/48x48x?`	directories of icons for *seemail*

BUGS

Edmail truncates long headers for searching.

NAME

man, lookman − print or find pages of this manual

SYNOPSIS

`man` [*option ...*] [*section section ...*] *title ...*

`lookman` *key ...*

DESCRIPTION

Man locates and prints pages of this manual named *title* in the specified *sections*. *Title* is given in lower case. Each *section* is a number; pages marked (2S), for example, belong to chapter 2. If no *section* is specified, pages in all sections are printed. Any name from the NAME section at the top of the page will serve as a *title*.

The options are:

`-p`	Run *proof*(1) on the specified man pages.
`-t`	Send the *troff* output to standard output.
`-n`	(Default) Print the pages on the standard output using *nroff*.

`Lookman` prints the names of all manual sections that contain all of the *key* words given on the command line.

FILES

`/sys/man/?/*`	*troff* source for manual; this page is `/sys/man/1/man`
`/sys/man/?/INDEX`	indices searched to find pages corresponding to titles
`/sys/lib/man/secindex`	command to make an index for a given section
`/sys/lib/man/lookman/index`	index for *lookman*

SEE ALSO

proof(1)

BUGS

The manual was intended to be typeset; some detail is sacrificed on text terminals.
There is no automatic mechanism to keep the indices up to date.
Except for special cases, it doesn't recognize things that should be run through *tbl* and/or *eqn*.

NAME

mc − multicolumn print

SYNOPSIS

`mc` [−] [−*N*] [`-t`] [*file* ...]

DESCRIPTION

Mc splits the input into as many columns as will fit in *N* print positions. If run in an *8½*(1) window, the default *N* is the number of blanks that will fit across the window; otherwise the default *N* is 80. Under option − each input line ending in a colon **:** is printed separately. On output, multiple spaces are converted to tabs; this is suppressed by option `-t`.

SEE ALSO

8½(1), *pr*(1)

NAME

mk, membername − maintain (make) related files

SYNOPSIS

`mk` [`-f` *mkfile*] ... [*option* ...] [*target* ...]

`membername` *aggregate* ...

DESCRIPTION

Mk uses the dependency rules specified in *mkfile* to control the update (usually by compilation) of *targets* (usually files) from the source files upon which they depend. The *mkfile* (default `mkfile`) contains a *rule* for each target that identifies the files and other targets upon which it depends and an *rc*(1) script, a *recipe*, to update the target. The script is run if the target does not exist or if it is older than any of the files it depends on. *Mkfile* may also contain *meta-rules* that define actions for updating implicit targets. If no *target* is specified, the target of the first rule (not meta-rule) in *mkfile* is updated.

The environment variable `$NPROC` determines how many targets may be updated simultaneously; Plan 9 sets `$NPROC` automatically to the number of CPUs on the current machine.

Options are:

`-a`	Assume all targets to be out of date. Thus, everything is updated.
`-d[egp]`	Produce debugging output (`p` is for parsing, `g` for graph building, `e` for execution).
`-e`	Explain why each target is made.
`-i`	Force any missing intermediate targets to be made.
`-k`	Do as much work as possible in the face of errors.
`-n`	Print, but do not execute, the commands needed to update the targets.
`-s`	Make the command line arguments sequentially rather than in parallel.
`-t`	Touch (update the modified date of) file targets, without executing any recipes.
`-w`*target1,target2,...*	Pretend the modify time for each *target* is the current time; useful in conjunction with `-n` to learn what updates would be triggered by modifying the *targets.*

The *rc*(1) script *membername* extracts member names (see ‘Aggregates’ below) from its arguments.

The `mkfile`

A *mkfile* consists of *assignments* (described under ‘Environment’) and *rules*. A rule contains *targets* and a *tail.* A target is a literal string and is normally a file name. The tail contains zero or more *prerequisites* and an optional *recipe,* which is an `rc` script. Each line of the recipe must begin with white space. A rule takes the form

```
target: prereq1 prereq2
        rc recipe using prereq1, prereq2 to build target
```

When the recipe is executed, the first character on every line is elided.

After the colon on the target line, a rule may specify *attributes*, described below.

A *meta-rule* has a target of the form *A*`%`*B* where *A* and *B* are (possibly empty) strings. A meta-rule acts as a rule for any potential target whose name matches *A*`%`*B* with `%` replaced by an arbitrary string, called the *stem*. In interpreting a meta-rule, the stem is substituted for all occurrences of `%` in the prerequisite names. In the recipe of a meta-rule, the environment variable `$stem` contains the string matched by the `%`. For example, a meta-rule to compile a C program using *2c*(1) might be:

```
%.2:    %.c
        2c $stem.c
        2l -o $stem $stem.2
```

Meta-rules may contain an ampersand `&` rather than a percent sign `%`. A `%` matches a maximal length string of any characters; an `&` matches a maximal length string of any characters except period or slash.

The text of the *mkfile* is processed as follows. Lines beginning with < followed by a file name are replaced by the contents of the named file. Blank lines and comments, which run from unquoted # characters to the following newline, are deleted. The character sequence backslash-newline is deleted, so long lines in *mkfile* may be folded. Non-recipe lines are processed by substituting for `` `{``*command*`}` the output of the *command* when run by *rc*. References to variables are replaced by the variables' values. Special characters may be quoted using single quotes `''` as in *rc*(1).

Assignments and rules are distinguished by the first unquoted occurrence of **:** (rule) or = (assignment).

A later rule may modify or override an existing rule under the following conditions:

- If the targets of the rules exactly match and one rule contains only a prerequisite clause and no recipe, the clause is added to the prerequisites of the other rule. If either or both targets are virtual, the recipe is always executed.
- If the targets of the rules match exactly and the prerequisites do not match and both rules contain recipes, *mk* reports an ``ambiguous recipe'' error.
- If the target and prerequisites of both rules match exactly, the second rule overrides the first.

Environment

Rules may make use of `rc` environment variables. A legal reference of the form `$OBJ` or `${name}` is expanded as in *rc*(1). A reference of the form `${name:`*A*`%`*B*`=`*C*`%`*D*`}`, where *A, B, C, D* are (possibly empty) strings, has the value formed by expanding `$name` and substituting *C* for *A* and *D* for *B* in each word in `$name` that matches pattern *A*`%`*B* **.**

Variables can be set by assignments of the form
var=[*attr*=]*value*
Blanks in the *value* break it into words, as in *rc* but without the surrounding parentheses. Such variables are exported to the environment of recipes as they are executed, unless `U`, the only legal attribute *attr*, is present. The initial value of a variable is taken from (in increasing order of precedence) the default values below, *mk's* environment, the *mkfiles*, and any command line assignment as an argument to *mk.* A variable assignment argument overrides the first (but not any subsequent) assignment to that variable. The variable `MKFLAGS` contains all the option arguments (arguments starting with − or containing =) and `MKARGS` contains all the targets in the call to *mk*.

It is recommended that mkfiles start with

```
</$objtype/mkfile
```

to set `CC`, `LD`, `RL`, `AS`, `O`, `LEX`, `YACC`, and `MK` to values appropriate to the target architecture (see the examples below).

Execution

During execution, *mk* determines which targets must be updated, and in what order, to build the *names* specified on the command line. It then runs the associated recipes.

A target is considered up to date if it has no prerequisites or if all its prerequisites are up to date and it is newer than all its prerequisites. Once the recipe for a target has executed, the target is considered up to date.

The date stamp used to determine if a target is up to date is computed differently for different types of targets. If a target is *virtual* (the target of a rule with the `V` attribute), its date stamp is initially zero; when the target is updated the date stamp is set to the most recent date stamp of its prerequisites. Otherwise, if a target does not exist as a file, its date stamp is set to the most recent date stamp of its prerequisites, or zero if it has no prerequisites. Otherwise, the target is the name of a file and the target's date stamp is always that file's modification date. The date stamp is computed when the target is needed in the execution of a rule; it is not a static value.

Nonexistent targets that have prerequisites and are themselves prerequisites are treated specially. Such a target *t* is given the date stamp of its most recent prerequisite and if this causes all the targets which have *t* as a prerequisite to be up to date, *t* is considered up to date. Otherwise, *t* is made in the normal fashion.

The `-i` flag overrides this special treatment.

Files may be made in any order that respects the preceding restrictions.

A recipe is executed by supplying the recipe as standard input to the command

```
/bin/rc -e -I
```

(the `-e` is omitted if the `E` attribute is set). The environment is augmented by the following variables:

`$alltarget`	all the targets of this rule.
`$newprereq`	the prerequisites that caused this rule to execute.
`$nproc`	the process slot for this recipe. It satisfies 0≤`$nproc`<`$NPROC`.
`$pid`	the process id for the *mk* executing the recipe.
`$prereq`	all the prerequisites for this rule.
`$stem`	if this is a meta-rule, `$stem` is the string that matched `%` or `&`. Otherwise, it is empty. For regular expression meta-rules (see below), the variables `stem0`, ..., `stem9` are set to the corresponding subexpressions.
`$target`	the targets for this rule that need to be remade.

These variables are available only during the execution of a recipe, not while evaluating the *mkfile.*

Unless the rule has the `Q` attribute, the recipe is printed prior to execution with recognizable environment variables expanded. Commands returning nonempty status (see *intro*(1)) cause *mk* to terminate.

Recipes and backquoted `rc` commands in places such as assignments execute in a copy of *mk's* environment; changes they make to environment variables are not visible from *mk.*

Variable substitution in a rule is done when the rule is read; variable substitution in the recipe is done when the recipe is executed. For example:

```
bar=a.c
foo:  $bar
        $CC -o foo $bar
bar=b.c
```

will compile `b.c` into `foo`, if `a.c` is newer than `foo`.

Aggregates

Names of the form *a*(*b*) refer to member *b* of the aggregate *a*. Currently, the only aggregates supported are *ar*(1) archives.

Attributes

The colon separating the target from the prerequisites may be immediately followed by *attributes* and another colon. The attributes are:

`<`	The standard output of the recipe is read by *mk* as an additional *mkfile.*
`D`	If the recipe exits with a non-null status, the target is deleted.
`E`	Continue execution if the recipe draws errors.
`N`	If there is no recipe, the target has its time updated.
`n`	The rule is a meta-rule that cannot be a target of a virtual rule. Only files match the pattern in the target.
`P`	The characters after the `P` until the terminating **:** are taken as a program name. It will be invoked as `rc -c prog 'arg1' 'arg2'` and should return a null exit status if and only if arg1 is not out of date with respect to arg2. Date stamps are still propagated in the normal way.
`Q`	The recipe is not printed prior to execution.
`R`	The rule is a meta-rule using regular expressions. In the rule, `%` has no special meaning. The target is interpreted as a regular expression as defined in *regexp*(6). The prerequisites may contain references to subexpressions in form *n,* as in the *source* argument to *regsub* (see *regexp*(2)).

`U`	The targets are considered to have been updated even if the recipe did not do so.
`V`	The targets of this rule are marked as virtual. They are distinct from files of the same name.

EXAMPLES

A simple mkfile to compile a program:

```
</$objtype/mkfile

prog:  a.$O b.$O c.$O
       $LD $CFLAGS -o $target $prereq

%.$O:  %.c
       $CC $stem.c
```

Override flag settings in the mkfile:

```
% mk target 'CFLAGS=-O -s'
```

To get the prerequisites for an aggregate:

```
% membername 'libc.a(read.2)' 'libc.a(write.2)'
read.2 write.2
```

Maintain a library:

```
libc.a(%.$O):N:     %.$O
libc.a:       libc.a(abs.$O) libc.a(access.$O) libc.a(alarm.$O) ...
       names=`{membername $newprereq}
       ar r libc.a $names && $RL libc.a && rm $names
```

String expression variables to derive names from a master list:

```
NAMES=alloc arc bquote builtins expand main match mk var word
OBJ=${NAMES:%=%.$O}
```

Regular expression meta-rules:

```
([^/]*)/(.*)\.o:R:  \1/\2.c
       cd $stem1; $CC $CFLAGS $stem2.c
```

A correct way to deal with *yacc*(1) grammars. The file `lex.c` includes the file `x.tab.h` rather than `y.tab.h` in order to reflect changes in content, not just modification time.

```
lex.o: x.tab.h
x.tab.h:     y.tab.h
       cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:    gram.y
       $YACC -d gram.y
```

The above example could also use the `P` attribute for the `x.tab.h` rule:

```
x.tab.h:Pcmp -s:    y.tab.h
       cp y.tab.h x.tab.h
```

SEE ALSO

rc(1), *regexp*(2)

A. Hume, ‘*Mk*: a Successor to *Make*’, Unix Research System Programmer’s Manual, Volume 2

BUGS

Identical recipes for regular expression meta-rules only have one target.

Seemingly appropriate input like `CFLAGS=-DHZ=60` is parsed as an erroneous attribute; correct it by inserting a space after the first =.

The recipes printed by *mk* before being passed to *rc* for execution are sometimes erroneously expanded for printing. Don’t trust what’s printed; rely on what *rc* does.

NAME

mkdir − make a directory

SYNOPSIS

`mkdir` *dirname ...*

DESCRIPTION

Mkdir creates specified directories. It requires write permission in the parent directory.

SEE ALSO

rm(1)
cd in *rc*(1)

DIAGNOSTICS

Mkdir returns null exit status if all directories were successfully made. Otherwise it prints a diagnostic and returns `"error"` status.

NAME

movie − algorithm animation

SYNOPSIS

`movie` [`-d`] [*files ...*]

DESCRIPTION

Movie converts a ‘movie script’ into an internal representation, then displays it in a window. Or, if the input file is already in the internal representation format, the conversion step is skipped. The `-d` option says to leave the internal representation around in a file with the same name as the input file, with the suffix (if any) replaced by `.i`. If no file names are given, standard input is used. If more than one file name is given, they are each animated in turn.

Button 1 stops and starts the movie; button 2 adjusts view sizes and selects clicks; button 3 sets various parameters. The ‘new file’ option on the button 3 menu prompts for another file to display; that file must be in internal representation format.

Movie scripts

A movie consists of multiple independent views, each presented as a rectangular sub-window. If no `view` statements appear, there is a single implicit view `def.view`. Any text or geometrical object may be labeled with a name and colon. Labels and coordinates are local to views. A recurring label erases the previous object with that label.

Comments follow #; blank lines are ignored.

`text` *options x y string*

Text is centered and medium size by default; options: one of `center` `ljust` `rjust` `above` `below`, and one of `small` `medium` `big` `bigbig`. A leading quote is stripped from *string*, as is a trailing quote if a leading one is present.

`line` *options x1 y1 x2 y2*

Lines are solid by default; options: one of `fat` `fatfat` `dotted` `dashed` and one of `->` `<-` `<->`.

`box` *options xmin ymin xmax ymax*

A box may be `filled`.

`circle` *options x1 y1 radius*

Radius is measured in the *x* dimension. A circle may be `filled`.

`erase` *label*

Erase an object explicitly.

`clear` Erase all objects currently in the current view.

`click` *optional-name*

Place a mark in the intermediate with this name; clicks are used to control stepping in a movie or to define frames for a set of stills.

`view` *name*

Associate subsequent objects with this view, until changed again.

FILES

All files are in `/sys/lib/movie/$objtype`.

`fdevelop`	Converts scripts to internal format.
`anim`	Displays one file in internal format.

SEE ALSO

J. L. Bentley and B. W. Kernighan, ‘A System for Algorithm Animation’, *Unix Programmer’s Manual,* Tenth Edition, Volume 2

NAME

netstat − summarize network connections

SYNOPSIS

`netstat`

DESCRIPTION

Netstat prints information about network connections. For *Datakit* connections *netstat* reports the connection number, the local user, the connection state, the service, and the address of the remote machine. For *IP* connections *netstat* reports the connection number, user, connection state, local port, remote port and remote address. *Netstat* looks up port numbers and addresses in the network databases to print symbolic names if possible.

FILES

`/net/*/*`

SEE ALSO

dkconfig(8), *ipconfig*(8)

NAME

news − print news items

SYNOPSIS

`news` [`-a`] [`-n`] [*item* ...]

DESCRIPTION

When invoked without options, this simple local news service prints files that have appeared in `/lib/news` since last reading, most recent first, with each preceded by an appropriate header. The time of reading is recorded. The options are

`-a` Print all items, regardless of currency. The recorded time is not changed.

`-n` Report the names of the current items without printing their contents, and without changing the recorded time.

Other arguments select particular news items.

To post a news item, create a file in `/lib/news`.

You may arrange to receive news automatically by registering your mail address in `/sys/lib/subscribers`. A daemon mails sends recent news to all addresses on the list.

FILES

`/lib/news/*`	articles
`$HOME/lib/newstime`	modify times is time of last read news
`/sys/lib/subscribers`	who gets news mailed to them

NAME

nm − name list (symbol table)

SYNOPSIS

`nm` [`-aghnsu`] *file* ...

DESCRIPTION

Nm prints the name list of each executable or object *file* in the argument list. If the *file* is an archive (see *ar*(1)), the name list of each file in the archive is printed. If more than one file is given in the argument list, the name of each file is printed at the beginning of each line.

Each symbol name is preceded by its hexadecimal value (blanks if undefined) and one of the letters

`T`	text segment symbol
`t`	static text segment symbol
`L`	leaf function text segment symbol
`l`	static leaf function text segment symbol
`D`	data segment symbol
`d`	static data segment symbol
`B`	bss segment symbol
`b`	static bss segment symbol
`a`	automatic (local) variable symbol
`p`	function parameter symbol
`z`	source file name
`Z`	source file line offset
`f`	source file name components

The output is sorted alphabetically.

Options are:

`-a`	Print all symbols; normally only user-defined text, data, and bss segment symbols are printed.
`-g`	Print only global (`T`, `L`, `D`, `B`) symbols.
`-h`	Do not print file name headers with output lines.
`-n`	Sort according to the address of the symbols.
`-s`	Don't sort; print in symbol-table order.
`-u`	Print only undefined symbols.

SEE ALSO

ar(1), *2l*(1), *db*(1)

NAME

p – paginate

SYNOPSIS

`p` [`–`*number*] [*file ...*]

DESCRIPTION

P copies its standard input, or the named files if given, to its standard output, stopping at the end of every 22nd line, and between files, to wait for a newline from the user. The page size may be set by saying (for example)

```
p -60
```

While waiting for a newline, *p* interprets the commands:

`!`	Pass the rest of the line to the shell as a command.
`q`	Quit.

NAME

passwd, typepasswd, netkey – change login password

SYNOPSIS

`passwd`

`aux/typepasswd`

`aux/netkey`

DESCRIPTION

Passwd changes your Plan 9 password. The program prompts for the old password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes. New passwords must be at least five characters long and be sufficiently hard to guess; passwords are not limited to eight characters. If the command is successful, the key in the encryption device is changed. Since each name space group has a separate encryption device, keys in other name space groups are not updated. For example, other windows will not have the new password. To correct the problem, reboot or run *typepasswd* in the other windows.

Typepasswd changes the key in the user's encryption device without changing the Plan 9 password.

Netkey uses the password to encrypt network challenges. It is a substitute for a SecureNet box.

These commands should be run only on a terminal; otherwise the password will be transmitted over a network in clear text.

FILES

`/dev/key`

SEE ALSO

encrypt(2), *cons*(3), *securenet*(8)

Robert Morris and Ken Thompson, 'UNIX password security,' *AT&T Bell Laboratories Technical Journal* 63 (1984) 1649-1672

NAME

pcc − APE C compiler driver

SYNOPSIS

`pcc` [*option ...*] [*name ...*]

DESCRIPTION

Pcc compiles and loads C programs, using APE (ANSI C/POSIX) include files and libraries. Named files ending with `.c` are preprocessed with *cpp*(1), then compiled with one of the compilers described in *2c*(1), as specified by the environment variable `$objtype`. The object files are then loaded using one of the loaders described in *2l*(1). The options are:

`-o` *out*	Place loader output in file *out* instead of the default `2.out`, `v.out`, etc.
`-P`	Omit the compilation and loading phases; leave the result of preprocessing *name*`.c` in *name*`.i`.
`-c`	Omit the loading phase.
`-p`	Insert profiling code into the executable output.
`-w`	Print compiler warning messages.
`-B`	Don't complain about functions used without ANSI function prototypes.
`-v`	Echo the preprocessing, compiling, and loading commands before they are executed.
`-D`*name=def* `-D`*name*	Define the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as `1`.
`-U`*name*	Undefine the *name* to the preprocessor, as if by `#undef`.
`-I`*dir*	`#include` files whose names do not begin with `/` are always sought first in the directory of the *file* argument, then in directories named in `-I` options, then in `/$objtype/include/ape`.
`-N`	Don't optimize compiled code.
`-S`	Print an assembly language version of the object code on standard output.
`-s`*name*	Print on standard output a listing of the fields in structure or union *name* together with their offsets and some type information. This can be used in conjunction with the debugger (see *db*(1)).

The APE environment contains all of the include files and library routines specified in the ANSI C standard (X3.159-1989), as well as those specified in the IEEE Portable Operating System Interface standard (POSIX, 1003.1-1990, ISO 9945-1). In order to access the POSIX routines, source programs should define the preprocessor constant `_POSIX_SOURCE`.

FILES

`/sys/include/ape`	system area for machine-independent `#include` directives.
`/$objtype/include/ape`	system area for machine-dependent `#include` directives.
`/$objtype/lib/ape/libap.a`	ANSI C/POSIX library.

SEE ALSO

cpp(1), *2c*(1), *2a*(1), *2l*(1), *rl*(1), *mk*(1), *nm*(1), *db*(1), *prof*(1)

Howard Trickey, *''APE — The ANSI/POSIX Environment''*

BUGS

The locale manipulation functions are minimal. Signal functions and terminal characteristic handlers are only minimally implemented. *Link* always fails, because Plan 9 doesn't support multiple links to a file. The functions related to setting effective user and group ids cannot be implemented because the concept doesn't exist in Plan 9.

NAME

pic, tpic − troff and tex preprocessors for drawing pictures

SYNOPSIS

`pic` [*files*]

`tpic` [*files*]

DESCRIPTION

Pic is a *troff*(1) preprocessor for drawing figures on a typesetter. *Pic* code is contained between `.PS` and `.PE` lines:

`.PS` *optional-width optional-height*
element-list
`.PE`

or in a file mentioned in a `.PS` line:

`.PS` *optional-width optional-height <file*

If *optional-width* is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion unless *optional-height* is present. If `.PF` is used instead of `.PE`, the typesetting position after printing is restored to what it was upon entry.

An *element-list* is a list of elements:

primitive attribute-list
placename **:** *element*
placename **:** *position*
var `=` *expr*
direction
`{` *element-list* `}`
`[` *element-list* `]`
`for` *var* `=` *expr* `to` *expr* `by` *expr* `do {` *anything* `}`
`if` *expr* `then {` *anything* `} else {` *anything* `}`
`copy` *file,* `copy thru` *macro,* `copy` *file* `thru` *macro*
`sh {` *commandline* `}`
`print` *expr*
`reset` *optional var-list*
troff-command

Elements are separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a `#` and terminated by a newline. Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

After each primitive the current position moves in the current direction (`up`,`down`, `left`,`right` (default)) by the size of the primitive. The current position and direction are saved upon entry to a `{`...`}` block and restored upon exit. Elements within a block enclosed in `[`...`]` are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block are local to that block.

Troff-command is any line that begins with a period. Such a line is assumed to make sense in the context where it appears; generally, this means only size and font changes.

The *primitive* objects are:

`box circle ellipse arc line arrow spline move` *text-list*

`arrow` is a synonym for `line ->`.

An *attribute-list* is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value.

`h(eigh)t` *expr* `wid(th)` *expr*

`rad(ius)` *expr*	`diam(eter)` *expr*
`up` *opt-expr*	`down` *opt-expr*
`right` *opt-expr*	`left` *opt-expr*
`from` *position*	`to` *position*
`at` *position*	`with` *corner*
`by` *expr, expr*	`then`
`dotted` *opt-expr*	`dashed` *opt-expr*
`chop` *opt-expr*	`->` `<-` `<->`
`invis`	`same`
text-list	*expr*

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. The attribute `at` causes the geometrical center to be put at the specified place; `with` causes the position on the object to be put at the specified place. For lines, splines and arcs, `height` and `width` refer to arrowhead size. A bare *expr* implies motion in the current direction.

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A text list is a list of text items:

text-item:
`"..."` *positioning ...*
`sprintf("`*format*`",` *expr*`,` *...*`)` *positioning ...*

positioning:
`center   ljust   rjust   above   below`

If there are multiple text items for some primitive, they are arranged vertically and centered except as qualified. Positioning requests apply to each item independently. Text items may contain *troff* commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:
expr, expr
place ± *expr, expr*
place ± `(` *expr, expr* `)`
`(` *position*`,` *position* `)`   *x* from one, *y* the other
expr [*of the way*] between *position* and *position*
expr < *position* , *position* >
(*position*)

place:
placename optional-corner
corner `of` *placename*
nth primitive optional-corner
corner `of` *nth primitive*
`Here`

An *optional-corner* is one of the eight compass points or the center or the start or end of a primitive.

optional-corner:
`.n   .e   .w   .s   .ne   .se   .nw   .sw   .c   .start   .end`

corner:
`top   bot   left   right   start   end`

Each object in a picture has an ordinal number; *nth* refers to this.

nth:
n`th`, *n*`th last`

The built-in variables and their default values are:

`boxwid 0.75`	`boxht 0.5`
`circlerad 0.25`	`arcrad 0.25`
`ellipsewid 0.75`	`ellipseht 0.5`

```
linewid 0.5            lineht 0.5
movewid 0.5            moveht 0.5
textwid 0              textht 0
arrowwid 0.05          arrowht 0.1
dashwid 0.1            arrowhead 2
scale 1
```

These may be changed at any time, and the new values remain in force from picture to picture until changed again or reset by a `reset` statement. Variables changed within `[` and `]` revert to their previous value upon exit from the block. Dimensions are divided by `scale` during output.

Expressions in *pic* are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expr:

```
expr op expr
- expr
! expr
( expr )
variable
number
place .x   place .y   place .ht   place .wid   place .rad
sin(expr)   cos(expr)   atan2(expr,expr)   log(expr)   exp(expr)
sqrt(expr)   max(expr,expr)   min(expr,expr)   int(expr)   rand()
```

op:

```
+  -  *  /  %  <  <=  >  >=  ==  !=  &&  ||
```

The `define` and `undef` statements are not part of the grammar.

```
define name { replacement text }
undef name
```

Occurrences of `$1`, `$2`, etc., in the replacement text will be replaced by the corresponding arguments if *name* is invoked as

```
name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings. Replacement text may contain newlines. The `undef` statement removes the definition of a macro.

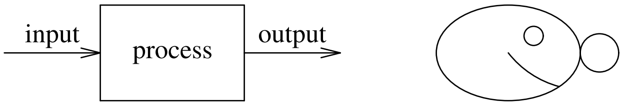
Tpic is a *tex*(1) preprocessor that accepts *pic* language. It produces Tex commands that define a box called `\graph`, which contains the picture. The box may be output this way:

```
\centerline{\box\graph}
```

EXAMPLES

```
arrow "input" above; box "process"; arrow "output" above
move
A: ellipse
   circle rad .1 with .w at A.e
   circle rad .05 at 0.5 <A.c, A.ne>
   arc from A.c to A.se rad 0.5
```

(. , .)... 0.000i 0.500i 3.200i 0.000i



SEE ALSO

grap(1), *doctype*(1), *troff*(1)

B. W. Kernighan, ‘PIC—a Graphics Language for Typesetting’, *Unix Programmer’s Manual,* Tenth Edition, Volume 2

NAME

plot − graphics filter

SYNOPSIS

`plot` [*file* ...]

DESCRIPTION

Plot interprets plotting instructions (see *plot*(6)) from the *files* or standard input, drawing the results on the screen, or in the current *8½*(1) window. Various options may be interspersed with the *file* arguments; they take effect at the given point in processing. Options are:

`-d`	Double buffer: make a copy off screen before writing the screen.
`-e`	Erase the screen.
`-c` *col*	Set the foreground color (see *plot*(6) for color names).
`-f` *fill*	Set the background color.
`-g` *grade*	Set the quality factor for arcs. Higher grades give better quality.
`-C`	Close the current plot.

SEE ALSO

8½(1), *plot*(6)

NAME

pr − print file

SYNOPSIS

`pr` [*option ...*] [*file ...*]

DESCRIPTION

Pr produces a printed listing of one or more *files.* The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. With no file arguments, *pr* prints its standard input.

Options apply to all following files but may be reset between files:

`-`*n*	Produce *n*-column output.
`+`*n*	Begin printing with page *n.*
`-b`	Balance columns on last page, in case of multi-column output.
`-d`	Double space.
`-h`	Take the next argument as a page header (*file* by default).
`-f`	Use formfeeds to separate pages.
`-l`*n*	Take the length of the page to be *n* lines instead of the default 66.
`-m`	Print all *files* simultaneously, each in one column.
`-n`	Number the lines of each *file.*
`-o`*n*	Offset the left margin *n* character positions.
`-s`*c*	Separate columns by the single character *c* instead of aligning them with white space. A missing *c* is taken to be a tab.
`-t`	Do not print the 5-line header or the 5-line trailer normally supplied for each page.
`-w`*n*	For purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72.

SEE ALSO

cat(1), *lp*(1)

NAME

prof, kprof − display profiling data

SYNOPSIS

`prof` [`-dr`] [*program*] [*profile*]

`kprof` *kernel kpdata*

DESCRIPTION

Prof interprets files produced automatically by programs loaded using the `-p` option of *2l* or appropriate loader. The symbol table in the named program file (`v.out` by default) is read and correlated with the profile file (`prof.out` by default). For each symbol, the percentage of time (in seconds) spent executing between that symbol and the next is printed (in decreasing order), together with the time spent there and the number of times that routine was called.

Under option `-d`, *prof* prints the dynamic call graph of the target program, annotating the calls with the time spent in each routine and those it calls, recursively. The output is indented two spaces for each call, and is formatted as

```
symbol:time/ncall
```

where *symbol* is the entry point of the call, *time* is in milliseconds, and *ncall* is the number of times that entry point was called at that point in the call graph. If *ncall* is one, the `/ncall` is elided. Normally recursive calls are compressed to keep the output brief; option `-r` prints the full call graph.

The size of the buffer in *program* used to hold the profiling data, by default 2000 entries, may be controlled by setting the environment variable `profsize` before running *program.* If the buffer fills, subsequent function calls may not be recorded.

Kprof is similar to *prof,* but presents the data accumulated by the kernel profiling device, *kprof*(3). The symbol table file, that of the operating system kernel, and the data file, typically `/dev/kpdata`, must be provided. *Kprof* has no options and cannot present dynamic data.

SEE ALSO

2l(1), *kprof*(3)

NAME

proof − troff output interpreter

SYNOPSIS

`proof` [`−m`*mag*] [`−/`*nview*] [`−F` *dir*] [`−d`] [*file*]

DESCRIPTION

Proof reads *troff*(1) intermediate language from *file* or standard input and simulates the resulting pages on the screen.

After a page of text is displayed, *proof* pauses for a command from the keyboard. The typed commands are:

newline	Go on to next page of text.
`−`	Go back to the previous page.
`q`	Quit.
`p`*n*	Print page *n*. An out-of-bounds page number means the end nearer to that number; a missing number means the current page; a signed number means an offset to the current page.
n	Same as `p`*n* .
`c`	Clear the screen, then wait for another command.
`m`*mag*	Change the magnification at which the output is printed. Normally it is printed with magnification .9; *mag*=.5 shrinks it to half size; *mag*=2 doubles the size.
`x`*val*	Move everything *val* screen pixels to the right (left, if *val* is negative).
`y`*val*	Move everything *val* screen pixels down (up, if *val* is negative).
`/`*nview*	Split the window into *nview* pieces. The current page goes into the rightmost, bottommost piece, and previous pages are shown in the other pieces.
`−F` *dir*	Use *dir* for fonts instead of `/lib/font/bit`.
`d`	Toggle the debug flag.

The `m`, `/`, and `d` commands are also available as command line options.

FILES

`/lib/font/bit/*`	fonts
`/lib/font/bit/MAP`	how to convert troff output fonts and character names into screen fonts and character numbers

SEE ALSO

lp(1), *psi*(1)
Brian W. Kernighan, *A Typesetter-independent Troff*

NAME

ps, psu − process status

SYNOPSIS

`ps`

`psu` [*user*]

DESCRIPTION

Ps prints information about processes. *Psu* prints only information about processes started by *user* (default `$user`).

For each process reported, the user, process id, user time, system time, size, state, and command name are printed. State is one of the following:

`Moribund`	Process has exited and is about to have its resources reclaimed.
`Ready`	on the queue of processes ready to be run.
`Scheding`	about to be run.
`Running`	running.
`Queueing`	waiting on a queue for a resource.
`Wakeme`	waiting for I/O or some other kernel event to wake it up.
`Broken`	dead of unnatural causes; lingering so that it can be examined.
`Stopped`	stopped.
`Stopwait`	waiting for another process to stop.
`Fault`	servicing a page fault.
`Idle`	waiting for something to do (kernel processes only).
`New`	being created.
`Pageout`	paging out some other process.
Syscall	performing the named system call.
`no` *resource*	waiting for more of a critical *resource.*

FILES

`/proc/*/status`

SEE ALSO

kill(1), *db*(1), *proc*(3)

NAME

psi − postscript interpreter

SYNOPSIS

`psi` [*option ...*] [*file*]

DESCRIPTION

Psi reads postscript input from *file* or from standard input and simulates the resulting pages in an *8½*(1) window. The options are

`-p`*n*	Display page *n.*
`-r`	Display the image at full scale, with the bottom left corner positioned at the bottom left corner of the window. (By default, the image is scaled to fit the window, maintaining the aspect ratio of a printer.)
`-a` *x y*	Display the image at full scale with position *x,y* of the image placed at the bottom left corner of the window.

Fonts are implemented with size-24 bitmap fonts. Available fonts are Symbol, Times-Roman, Times-Italic, Times-Bold, Helvetica, Helvetica-Oblique, Helvetica-Bold.

When the 'cherries' icon is displayed, use mouse button 3 to move forward (`more`) or quit (`done`). Button 2 exits the program completely.

EXAMPLE

```
troff -ms memo | lp -dstdout -H | psi
```

Format a memo, convert it to postscript, and display it.

FILES

`psi.err`	error messages

SEE ALSO

lp(1), *proof*(1)

DIAGNOSTICS

Error comments are placed on file `psi.err`.

Symbols that lack bitmaps are replaced by '?' and an error is reported.

BUGS

Unimplemented postscript features are rotated images or imagemasks, half tone screens, multiple path clipping, and charpath.

NAME

push, pull, Rpush, Rpull − Datakit remote file copy

SYNOPSIS

`push` [`-v`] *machine file ... remotedir*

`pull` [`-v`] *machine file ... localdir*

DESCRIPTION

Push and *pull* copy files between machines over Datakit. *Push* copies *files* from the local machine to the directory *remotedir* on the named *machine*. *Pull* copies *files* from the named *machine* to the directory *localdir* on the local machine. The last component of the name of a copy is the same as that of the original. If one of the *files* is a directory, a corresponding directory is created and the directory's files are copied, recursively.

Option `-v` announces each file as it is copied.

Pushing and pulling involve two programs running in different contexts on different machines. In particular, pulling to directory **.** puts files in the local current directory, but pushing to **.** puts files in the remote home directory. Shell metacharacters which are to be interpreted on the remote machine must be quoted.

Rpush and *Rpull* are the programs started by remote pushes and pulls.

SEE ALSO

con(1), *cp*(1)

DIAGNOSTICS

Messages marked `(remote)` are from the sister process running on the remote machine.

NAME

pwd, pbd − working directory

SYNOPSIS

`pwd`

`pbd`

DESCRIPTION

Pwd prints the path name of the working (current) directory. Pwd does not guarantee to return the same path that was used to enter the directory. The returned path may be another route through the name space to the same working directory. This behavior will arise when a combination of mounts or binds produces a graph in the file tree.

Pbd prints the base name of the working (current) directory. It prints no final newline and is intended for applications such as constructing shell prompts.

SEE ALSO

cd in *rc*(1), *bind*(1), *getwd*(2)

BUGS

A kernel bug sometimes prevents `pbd` from working when the current directory is a device, in which case *pbd* returns ‘.’.

NAME

rc, cd, eval, exec, exit, flag, newpgrp, shift, wait, whatis, . − command language

SYNOPSIS

`rc` [`-srdiIlxepvV`] [`-c command`] [*file* [*arg* ...]

DESCRIPTION

Rc is the Plan 9 shell. It executes command lines read from a terminal or a file or, with the `-c` flag, from *rc's* argument list.

Command Lines

A command line is a sequence of commands, separated by ampersands or semicolons (`&` or `;`) and terminated by a newline. The commands are executed in sequence from left to right. *Rc* does not wait for a command followed by `&` to finish executing before starting the following command. Whenever a command followed by `&` is executed, its process id is assigned to the *rc* variable `$apid`. Whenever a command *not* followed by `&` exits or is terminated, the *rc* variable `$status` gets the process's wait message (see *wait*(2)); it will be the null string if the command was successful.

A long command line may be continued on subsequent lines by typing a backslash (`\`) followed by a newline. This sequence is treated as though it were a blank. Backslash is not otherwise a special character.

A number-sign (`#`) and any following characters up to (but not including) the next newline are ignored, except in quotation marks.

Simple Commands

A simple command is a sequence of arguments interspersed with I/O redirections. If the first argument is the name of an *rc* function or of one of *rc's* built-in commands, it is executed by *rc*. Otherwise if the name starts with a slash (`/`), it must be the path name of the program to be executed. Names containing no initial slash are searched for in a list of directory names stored in `$path`. The first executable file of the given name found in a directory in `$path` is the program to be executed. To be executable, the user must have execute permission (see *stat*(2)) and the file must be either an executable binary for the current machine's CPU type, or a shell script. Shell scripts begin with a line containing the full path name of a shell (usually `/bin/rc`), prefixed by `#!`.

The first word of a simple command cannot be a keyword unless it is quoted or otherwise disguised. The keywords are

```
for in while if not switch fn ~ ! @
```

Arguments and Variables

A number of constructions may be used where *rc's* syntax requires an argument to appear. In many cases a construction's value will be a list of arguments rather than a single string.

The simplest kind of argument is the unquoted word: a sequence of one or more characters none of which is a blank, tab, newline, or any of the following:

```
# ; & | ^ $ = ` ' { } ( ) < >
```

An unquoted word that contains any of the characters `*` `?` `[` is a pattern for matching against file names. The character `*` matches any sequence of characters, `?` matches any single character, and `[`*class*`]` matches any character in the *class*. If the first character of *class* is `~`, the class is complemented. The *class* may also contain pairs of characters separated by `-`, standing for all characters lexically between the two. The character `/` must appear explicitly in a pattern, as must the first character of the path name components `.` and `..`. A pattern is replaced by a list of arguments, one for each path name matched, except that a pattern matching no names is not replaced by the empty list, but rather stands for itself. Pattern matching is done after all other operations. Thus,

```
x=/tmp echo $x^/*.c
```

matches `/tmp/*.c`, rather than matching `/*.c` and then prefixing `/tmp`.

A quoted word is a sequence of characters surrounded by single quotes (`'`). A single quote is represented in a quoted word by a pair of quotes (`''`).

Each of the following is an argument.

`(`*arguments*`)`

The value of a sequence of arguments enclosed in parentheses is a list comprising the members of each element of the sequence. Argument lists have no recursive structure, although their syntax may suggest it. The following are entirely equivalent:

```
echo hi there everybody
((echo) (hi there) everybody)
```

`$`*argument*

`$`*argument*`(`*subscript*`)`

The *argument* after the `$` is the name of a variable whose value is substituted. Multiple levels of indirection are possible, but of questionable utility. Variable values are lists of strings. If *argument* is a number *n*, the value is the *n*th element of `$*`, unless `$*` doesn't have *n* elements, in which case the value is empty. If *argument* is followed by a parenthesized list of subscripts, the value substituted is a list composed of the requested elements (origin 1). The parenthesis must follow the variable name with no spaces. Assignments to variables are described below.

`$#`*argument*

The value is the number of elements in the named variable. A variable never assigned a value has zero elements.

$"*argument*

The value is a single string containing the components of the named variable separated by spaces. A variable with zero elements yields the empty string.

`` `{ ``*command*`}`

rc executes the *command* and reads its standard output, splitting it into a list of arguments, using characters in `$ifs` as separators. If `$ifs` is not otherwise set, its value is `' \t\n'`.

`<{`*command*`}`

`>{`*command*`}`

The *command* is executed asynchronously with its standard output or standard input connected to a pipe. The value of the argument is the name of a file referring to the other end of the pipe. This allows the construction of non-linear pipelines. For example, the following runs two commands `old` and `new` and uses `cmp` to compare their outputs

```
cmp <{old} <{new}
```

argument`^`*argument*

The `^` operator concatenates its two operands. If the two operands have the same number of components, they are concatenated pairwise. If not, then one operand must have one component, and the other must be non-empty, and concatenation is distributive.

Free Carets

In most circumstances, *rc* will insert the `^` operator automatically between words that are not separated by white space. Whenever one of `` $ ' ` `` follows a quoted or unquoted word or an unquoted word follows a quoted word with no intervening blanks or tabs, a `^` is inserted between the two. If an unquoted word immediately follows a `$` and contains a character other than an alphanumeric, underscore, or `*`, a `^` is inserted before the first such character. Thus

```
cc -$flags $stem.c
```

is equivalent to

```
cc -^$flags $stem^.c
```

I/O Redirections

The sequence >*file* redirects the standard output file (file descriptor 1, normally the terminal) to the named *file;* >>*file* appends standard output to the file. The standard input file (file descriptor 0, also normally the terminal) may be redirected from a file by the sequence <*file*`,` or from an inline 'here document' by the sequence <<*eof-marker.* The contents of a here document are lines of text taken from the command input stream up to a line containing nothing but the *eof-marker,* which may be either a quoted or unquoted word. If *eof-marker* is unquoted, variable names of the form `$`*word* have their values substituted from *rc's* environment. If `$`*word* is followed by a caret (`^`), the caret is deleted. If *eof-marker* is quoted, no substitution occurs.

Redirections may be applied to a file-descriptor other than standard input or output by qualifying the redirection operator with a number in square brackets. For example, the diagnostic output (file descriptor 2) may be redirected by writing `cc junk.c >[2]junk`.

A file descriptor may be redirected to an already open descriptor by writing `>[`*fd0*`=`*fd1*`]` or `<[`*fd0*`=`*fd1*`]`. *Fd1* is a previously opened file descriptor and *fd0* becomes a new copy (in the sense of *dup*(2)) of it. A file descriptor may be closed by writing `>[`*fd0*`=]` or `<[`*fd0*`=]`.

Redirections are executed from left to right. Therefore, `cc junk.c >/dev/null >[2=1]` and `cc junk.c >[2=1] >/dev/null` have different effects – the first puts standard output in `/dev/null` and then puts diagnostic output in the same place, where the second directs diagnostic output to the terminal and sends standard output to `/dev/null`.

Compound Commands

A pair of commands separated by a pipe operator (`|`) is a command. The standard output of the left command is sent through a pipe to the standard input of the right command. The pipe operator may be decorated to use different file descriptors. `|[`*fd*`]` connects the output end of the pipe to file descriptor *fd* rather than 1. `|[`*fd0*`=`*fd1*`]` connects output to *fd0* of the left command and input to *fd1* of the right command.

A pair of commands separated by `&&` or `||` is a command. In either case, the left command is executed and its exit status examined. If the operator is `&&` the right command is executed if the left command's status is null. `||` causes the right command to be executed if the left command's status is non-null.

The exit status of a command may be inverted (non-null is changed to null, null is changed to non-null) by preceding it with a `!`.

The `|` operator has highest precedence, and is left-associative (i.e. binds tighter to the left than the right.) `!` has intermediate precedence, and `&&` and `||` have the lowest precedence.

The unary `@` operator, with precedence equal to `!`, causes its operand to be executed in a subshell.

Each of the following is a command.

`if (` *list* `)` *command*

A *list* is a sequence of commands, separated by `&`, `;`, or newline. It is executed and if its exit status is null, the *command* is executed.

`if not` *command*

The immediately preceding command must have been `if(`*list*`)` *command.* If its condition was non-zero, the *command* is executed.

`for(`*name* `in` *arguments*`)` *command*

`for(`*name*`)` *command*

The *command* is executed once for each *argument* with that argument assigned to *name*. If the argument list is omitted, `$*` is used.

`while(`*list*`)` *command*

The *list* is executed repeatedly until its exit status is non-null. Each time it returns null status, the *command* is executed. The empty *list* always yields zero status.

`switch(`*argument*`){`*list*`}`

The *list* is searched for simple commands beginning with the word `case`. (The search is only at the 'top level' of the *list*. That is, `cases` in nested constructs are not found.) *Argument* is matched against each word following `case` using the pattern-matching algorithm described above, except that `/` and the first characters of `.` and `..` need not be matched explicitly. When a match is found, commands in the list are executed up to the next following `case` command (at the top level) or the closing parenthesis.

`{`*list*`}`

Braces serve to alter the grouping of commands implied by operator priorities. The *body* is a sequence of commands separated by `&`, `;`, or newline.

`fn` *name* { *list* }
`fn` *name*
The first form defines a function with the given *name*. Subsequently, whenever a command whose first argument is *name* is encountered, the current value of the remainder of the command's argument list will be assigned to `$*`, after saving its current value, and *rc* will execute the *list*. The second form removes *name*'s function definition.

`fn` *note* { *list* }
`fn` *note*
A function with the name of a note is defined in the usual way, but called when *rc* receives that note; see *notify*(2). The valid note names are `sigexit`, `sighup`, `sigint`, `sigquit`, `sigalrm`, `sigbpt`, `sigrange`, `sigodd`, `sigbadsys`, `sigoddstack`, `sigpipe`, `sigtrap`, and `sigfpe`. By default *rc* exits on receiving any signal, except when run interactively, in which case interrupts and quits normally cause *rc* to stop whatever it's doing and start reading a new command. The second form causes *rc* to handle a signal in the default manner. *Rc* recognizes an artificial signal, `sigexit`, which occurs when *rc* is about to finish executing.

name=*argument command*
Any command may be preceded by a sequence of assignments interspersed with redirections. The assignments remain in effect until the end of the command, unless the command is empty (i.e. the assignments stand alone), in which case they are effective until rescinded by later assignments.

Built-in Commands

These commands are executed internally by *rc*, usually because their execution changes or depends on *rc*'s internal state.

`.` *file ...*
Execute commands from *file*. `$*` is set for the duration to the remainder of the argument list following *file*. *File* is searched for using `$path`.

`builtin` *command ...*
Execute *command* as usual except that any function named *command* is ignored.

`cd [`*dir*`]`
Change the current directory to *dir*. The default argument is `$home`. *dir* is searched for in each of the directories mentioned in `$cdpath`.

`eval [`*arg ...*`]`
The arguments are concatenated separated by spaces into a single string, read as input to *rc*, and executed.

`exec [`*command ...*`]`
This instance of *rc* replaces itself with the given (non-built-in) *command*.

`flag` *f* `[+-]`
Either set (+), clear (−), or test (neither + nor −) the flag *f*, where *f* is a single character, one of the command line flags (see Invocation, below).

`exit [`*status*`]`
Exit with the given exit status. If none is given, the current value of `$status` is used.

`rfork [nNeEsfF]`
Become a new process group using `rfork(`*flags*`)` where *flags* is composed of the bitwise OR of the `rfork` flags specified by the option letters (see *fork*(2)). If no *flags* are given, they default to `ens`. The *flags* and their meanings are: `n` is `RFNAMEG`; `N` is `RFCNAMEG`; `e` is `RFENVG`; `E` is `RFCENVG`; `s` is `RFNOTEG`; `f` is `RFFDG`; and `F` is `RFCFDG`.

`shift [`*n*`]`
Delete the first *n* (default 1) elements of `$*`.

`wait [`*pid*`]`
Wait for the process with the given *pid* to exit. If no *pid* is given, all outstanding processes are waited for.

`whatis` *name ...*
Print the value of each *name* in a form suitable for input to *rc*. The output is an assignment to any variable, the definition of any function, a call to `builtin` for any built-in command, or the

completed pathname of any executable file.

~ *subject pattern ...*

The *subject* is matched against each *pattern* in sequence. If it matches any pattern, `$status` is set to zero. Otherwise, `$status` is set to one. Patterns are the same as for file name matching, except that `/` and the first character of `.` and `..` need not be matched explicitly. The *patterns* are not subjected to file name matching before the `~` command is executed, so they need not be enclosed in quotation marks.

Environment

The *environment* is a list of strings made available to executing binaries by the `env` device (see *env*(3)). *Rc* creates an environment entry for each variable whose value is non-empty, and for each function. The string for a variable entry has the variable's name followed by = and its value. If the value has more than one component, these are separated by ctrl-a (`'\001'`) characters. The string for a function is just the *rc* input that defines the function. The name of a function in the environment is the function name preceded by `fn#`.

When *rc* starts executing it reads variable and function definitions from its environment.

Special Variables

The following variables are set or used by *rc*.

`$*`	Set to *rc*'s argument list during initialization. Whenever a `.` command or a function is executed, the current value is saved and `$*` receives the new argument list. The saved value is restored on completion of the `.` or function.
`$apid`	Whenever a process is started asynchronously with `&`, `$apid` is set to its process id.
`$home`	The default directory for `cd`.
`$ifs`	The input field separators used in backquote substitutions. If `$ifs` is not set in *rc*'s environment, it is initialized to blank, tab and newline.
`$path`	The search path used to find commands and input files for the `.` command. If not set in the environment, it is initialized by `path=(. /bin)`. Its use is discouraged; instead use *bind*(1) to build a `/bin` containing what's needed.
`$pid`	Set during initialization to *rc*'s process id.
`$prompt`	When *rc* is run interactively, the first component of `$prompt` is printed before reading each command. The second component is printed whenever a newline is typed and more lines are required to complete the command. If not set in the environment, it is initialized by `prompt=('% ' ' ')`.
`$status`	Set to the wait message of the last-executed program. (unless started with `&`). `!` and `~` also change `$status`. Its value is used to control execution in `&&`, `\|\|`, `if` and `while` commands. When *rc* exits at end-of-file of its input or on executing an `exit` command with no argument, `$status` is its exit status.

Invocation

If *rc* is started with no arguments it reads commands from standard input. Otherwise its first non-flag argument is the name of a file from which to read commands (but see `-c` below). Subsequent arguments become the initial value of `$*`. *Rc* accepts the following command-line flags.

`-c` *string*	Commands are read from *string*.
`-s`	Print out exit status after any command where the status is non-null.
`-e`	Exit if `$status` is non-null after executing a simple command.
`-i`	If `-i` is present, or *rc* is given no arguments and its standard input is a terminal, it runs interactively. Commands are prompted for using `$prompt`.
`-I`	Makes sure *rc* is not run interactively.
`-l`	If `-l` is given or the first character of argument zero is `-`, *rc* reads commands from `$home/lib/profile`, if it exists, before reading its normal input.
`-p`	A no-op.
`-d`	A no-op.
`-v`	Echo input on file descriptor 2 as it is read.

`-x` Print each simple command before executing it. `-r` Print debugging information (internal form of commands as they are executed).

BUGS

It’s too slow and too big.
There should be a way to match patterns against whole lists rather than just single strings.
Using ~ to check the value of `$status` changes `$status`.
Functions that use here documents don’t work.

NAME

rl − put table of contents in libraries

SYNOPSIS

`rl` [*-v*] [*name ...*]

DESCRIPTION

rl add tables of contents to the named files, which should be archives of one type of object files. The loaders (see *2l*(1)) need a table of contents in libraries.

The `−v` option prints debugging information (the names of the symbols, their type, and their offset in the archive).

SEE ALSO

ar(1), *2l*(1), *nm*(1)

NAME

rm − remove files

SYNOPSIS

`rm` [`-fr`] *file* ...

DESCRIPTION

Rm removes files or directories. A directory is removed only if it is empty. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself. The options are

`-f` Don't report files that can't be removed.

`-r` Recursively delete the entire contents of a directory and the directory itself.

SEE ALSO

remove(2)

NAME

sam, B − screen editor with structural regular expressions

SYNOPSIS

`sam` [*option ...*] [*files*]

`sam -r` *machine*

`sam.save`

`B` [`-nnnn`] *file ...*

DESCRIPTION

Sam is a multi-file editor. It modifies a local copy of an external file. The copy is here called a *file*. The files are listed in a menu available through mouse button 3 or the `n` command. Each file has an associated name, usually the name of the external file from which it was read, and a 'modified' bit that indicates whether the editor's file agrees with the external file. The external file is not read into the editor's file until it first becomes the current file—that to which editing commands apply—whereupon its menu entry is printed. The options are

`-d` Do not download the terminal part of *sam.* Editing will be done with the command language only, as in *ed*(1).

`-r` *machine*
Run the host part remotely on the specified machine, the terminal part locally.

`-s` *path*
Start the host part from the specified file on the remote host. Only meaningful with the `-r` option.

`-t` *path*
Start the terminal part from the specified file. Useful for debugging.

Regular expressions

Regular expressions are as in *regexp*(6) with the addition of `\n` to represent newlines. A regular expression may never contain a literal newline character. The empty regular expression stands for the last complete expression encountered. A regular expression in *sam* matches the longest leftmost substring formally matched by the expression. Searching in the reverse direction is equivalent to searching backwards with the catenation operations reversed in the expression.

Addresses

An address identifies a substring in a file. In the following, 'character *n*' means the null string after the *n*-th character in the file, with 1 the first character in the file. 'Line *n*' means the *n*-th match, starting at the beginning of the file, of the regular expression `.*\n?` . All files always have a current substring, called dot, that is the default address.

Simple Addresses

`#`*n* The empty string after character *n*; `#0` is the beginning of the file.

n Line *n*.

`/`*regexp*`/`
`?`*regexp*`?`
The substring that matches the regular expression, found by looking toward the end (`/`) or beginning (`?`) of the file, and if necessary continuing the search from the other end to the starting point of the search. The matched substring may straddle the starting point.

`0` The string before the first full line. This is not necessarily the null string; see + and − below.

`$` The null string at the end of the file.

`.` Dot.

`'` The mark in the file (see the `k` command below).

`"`*regexp*`"`
: Preceding a simple address (default `.`), refers to the address evaluated in the unique file whose menu line matches the regular expression.

Compound Addresses

In the following, *a1* and *a2* are addresses.

a1`+`*a2* The address *a2* evaluated starting at the end of *a1*.

a1`-`*a2* The address *a2* evaluated looking in the reverse direction starting at the beginning of *a1*.

a1`,`*a2* The substring from the beginning of *a1* to the end of *a2*. If *a1* is missing, `0` is substituted. If *a2* is missing, `$` is substituted.

a1`;`*a2* Like *a1*`,`*a2,* but with *a2* evaluated at the end of, and dot set to, *a1*.

The operators + and − are high precedence, while `,` and `;` are low precedence.

In both + and − forms, if *a2* is a line or character address with a missing number, the number defaults to 1. If *a1* is missing, `.` is substituted. If both *a1* and *a2* are present and distinguishable, + may be elided. *a2* may be a regular expression; if it is delimited by `?`'s, the effect of the + or − is reversed.

It is an error for a compound address to represent a malformed substring. Some useful idioms: *a1*`+-` (*a1*`-+`*)* selects the line containing the end (beginning) of a1. `0/`*regexp*`/` locates the first match of the expression in the file. (The form `0;//` sets dot unnecessarily.) `./`*regexp*`///` finds the second following occurrence of the expression, and `.,/`*regexp*`/` extends dot.

Commands

In the following, text demarcated by slashes represents text delimited by any printable character except alphanumerics. Any number of trailing delimiters may be elided, with multiple elisions then representing null strings, but the first delimiter must always be present. In any delimited text, newline may not appear literally; `\n` may be typed for newline; and `\/` quotes the delimiter, here `/`. Backslash is otherwise interpreted literally, except in `s` commands.

Most commands may be prefixed by an address to indicate their range of operation. Those that may not are marked with a `*` below. If a command takes an address and none is supplied, dot is used. The sole exception is the `w` command, which defaults to `0,$`. In the description, 'range' is used to represent whatever address is supplied. Many commands set the value of dot as a side effect. If so, it is always set to the 'result' of the change: the empty string for a deletion, the new text for an insertion, etc. (but see the `s` and `e` commands).

Text commands

`a/`*text*`/`
or
`a`
lines of text
`.` Insert the text into the file after the range. Set dot.

`c`
`i` Same as `a`, but `c` replaces the text, while `i` inserts *before* the range.

`d` Delete the text in the range. Set dot.

`s/`*regexp*`/`*text*`/`
: Substitute *text* for the first match to the regular expression in the range. Set dot to the modified range. In *text* the character `&` stands for the string that matched the expression. Backslash behaves as usual unless followed by a digit: `\`*d* stands for the string that matched the subexpression begun by the *d*-th left parenthesis. If *s* is followed immediately by a number *n*, as in `s2/x/y/`, the *n*-th match in the range is substituted. If the command is followed by a `g`, as in `s/x/y/g`, all matches in the range are substituted.

`m` *a1*
`t` *a1* Move the range to after *a1* (`m`), or copy it (`t`). Set dot.

Display commands

`p` Print the text in the range. Set dot.

`=` Print the line address and character address of the range.

`=#` Print just the character address of the range.

File commands

* `b` *file-list*

Set the current file to the first file named in the list that *sam* also has in its menu. The list may be expressed `<`*Plan 9 command* in which case the file names are taken as words (in the shell sense) generated by the Plan 9 command.

* `B` *file-list*

Same as `b`, except that file names not in the menu are entered there, and all file names in the list are examined.

* `n` Print a menu of files. The format is:

`'` or blank

indicating the file is modified or clean,

`-` or `+` indicating the file is unread or has been read (in the terminal, `*` means more than one window is open),

`.` or blank

indicating the current file,

a blank,

and the file name.

* `D` *file-list*

Delete the named files from the menu. If no files are named, the current file is deleted. It is an error to `D` a modified file, but a subsequent `D` will delete such a file.

I/O Commands

* `e` *filename*

Replace the file by the contents of the named external file. Set dot to the beginning of the file.

`r` *filename*

Replace the text in the range by the contents of the named external file. Set dot.

`w` *filename*

Write the range (default `0,$`) to the named external file.

* `f` *filename*

Set the file name and print the resulting menu entry.

If the file name is absent from any of these, the current file name is used. `e` always sets the file name, `r` and `w` do so if the file has no name.

`<` *Plan 9-command*

Replace the range by the standard output of the Plan 9 command.

`>` *Plan 9-command*

Sends the range to the standard input of the Plan 9 command.

`|` *Plan 9-command*

Send the range to the standard input, and replace it by the standard output, of the Plan 9 command.

* `!` *Plan 9-command*

Run the Plan 9 command.

* `cd` *directory*

Change working directory. If no directory is specified, `$home` is used.

In any of `<`, `>`, `|` or `!`, if the *Plan 9 command* is omitted the last *Plan 9 command* (of any type) is substituted. If *sam* is downloaded, `!` sets standard input to `/dev/null`, and otherwise unassigned output (`stdout` for `!` and `>`, `stderr` for all) is placed in `/tmp/sam.err` and the first few lines are printed.

Loops and Conditionals

`x/`*regexp*`/` *command*

For each match of the regular expression in the range, run the command with dot set to the match. Set dot to the last match. If the regular expression and its slashes are omitted, `/.*\n/` is assumed. Null string matches potentially occur before every character of the range and at the end of the range.

`y/` *regexp* `/` *command*
: Like `x,` but run the command for each substring that lies before, between, or after the matches that would be generated by `x`. There is no default behavior. Null substrings potentially occur before every character in the range.

* `X/` *regexp* `/` *command*
: For each file whose menu entry matches the regular expression, make that the current file and run the command. If the expression is omitted, the command is run in every file.

* `Y/` *regexp* `/` *command*
: Same as `X`, but for files that do not match the regular expression, and the expression is required.

`g/` *regexp* `/` *command*

`v/` *regexp* `/` *command*
: If the range contains (`g`) or does not contain (`v`) a match for the expression, set dot to the range and run the command.

These may be nested arbitrarily deeply, but only one instance of either `X` or `Y` may appear in a single command. An empty command in an `x` or `y` defaults to `p`; an empty command in `X` or `Y` defaults to `f.` `g` and `v` do not have defaults.

Miscellany

`k` Set the current file's mark to the range. Does not set dot.

* `q` Quit. It is an error to quit with modified files, but a second `q` will succeed.

* `u` *n* Undo the last *n* (default 1) top-level commands that changed the contents or name of the current file, and any other file whose most recent change was simultaneous with the current file's change. Successive `u`'s move further back in time. The only commands for which u is ineffective are `cd`, `u`, `q`, `w` and `D`.

(empty) If the range is explicit, set dot to the range. If *sam* is downloaded, the resulting dot is selected on the screen; otherwise it is printed. If no address is specified (the command is a newline) dot is extended in either direction to line boundaries and printed. If dot is thereby unchanged, it is set to `.+1` and printed.

Grouping and multiple changes

Commands may be grouped by enclosing them in braces `{}`. Commands within the braces must appear on separate lines (no backslashes are required between commands). Semantically, an opening brace is like a command: it takes an (optional) address and sets dot for each sub-command. Commands within the braces are executed sequentially, but changes made by one command are not visible to other commands (see the next paragraph). Braces may be nested arbitrarily.

When a command makes a number of changes to a file, as in `x/re/c/text/`, the addresses of all changes to the file are computed in the original file. If the changes are in sequence, they are applied to the file. Successive insertions at the same address are catenated into a single insertion composed of the several insertions in the order applied.

The terminal

What follows refers to behavior of *sam* when downloaded, that is, when operating as a display editor on a bitmap display. This is the default behavior; invoking *sam* with the `-d` (no download) option provides access to the command language only.

Each file may have zero or more windows open. Each window is equivalent and is updated simultaneously with changes in other windows on the same file. Each window has an independent value of dot, indicated by a highlighted substring on the display. Dot may be in a region not within the window. There is usually a 'current window', marked with a dark border, to which typed text and editing commands apply. Text may be typed and edited as in *8½*(1); also the escape key (ESC) selects (sets dot to) text typed since the last mouse button hit.

The button 3 menu controls window operations. The top of the menu provides the following operators, each of which uses one or more *8½*-like cursors to prompt for selection of a window or sweeping of a rectangle. 'Sweeping' a null rectangle gets a large window, disjoint from the command window or the whole screen, depending on where the null rectangle is.

`new`	Create a new, empty file.
`xerox`	Create a copy of an existing window.
`reshape`	As in *8½*.
`close`	Delete the window. In the last window of a file, `close` is equivalent to a `D` for the file.
`write`	Equivalent to a `w` for the file.

Below these operators is a list of available files, starting with `~~sam~~`, the command window. Selecting a file from the list makes the most recently used window on that file current, unless it is already current, in which case selections cycle through the open windows. If no windows are open on the file, the user is prompted to open one. Files other than `~~sam~~` are marked with one of the characters `-+*` according as zero, one, or more windows are open on the file. A further mark `.` appears on the file in the current window and a single quote, `'`, on a file modified since last write.

The command window, created automatically when `sam` starts, is an ordinary window except that text typed to it is interpreted as commands for the editor rather than passive text, and text printed by editor commands appears in it. The behavior is like *8½,* with an 'output point' that separates commands being typed from previous output. Commands typed in the command window apply to the current open file—the file in the most recently current window.

Manipulating text

Button 1 changes selection, much like *8½*. Pointing to a non-current window with button 1 makes it current; within the current window, button 1 selects text, thus setting dot. Double-clicking selects text to the boundaries of words, lines, quoted strings or bracketed strings, depending on the text at the click.

Button 2 provides a menu of editing commands:

`cut`	Delete dot and save the deleted text in the snarf buffer.
`paste`	Replace the text in dot by the contents of the snarf buffer.
`snarf`	Save the text in dot in the snarf buffer.
`look`	Search forward for the next occurrence of the literal text in dot. If dot is the null string, the text in the snarf buffer is used. The snarf buffer is unaffected.
`<8½>`	Exchange snarf buffers with *8½*.
/ *regexp*	Search forward for the next match of the last regular expression typed in a command. (Not in command window.)
`send`	Send the text in dot, or the snarf buffer if dot is the null string, as if it were typed to the command window. Saves the sent text in the snarf buffer. (Command window only.)

External communication

On invocation *sam* creates a named pipe `/srv/sam.`*user* which acts as an additional source of commands. Characters written to the named pipe are treated as if they had been typed in the command window. This is usually used to issue `B` commands from the shell.

B is a shell-level command that causes an instance of *sam* running on the same terminal to load the named *files.* The option allows a line number to be specified for the initial position to display in the last named file.

Abnormal termination

If *sam* terminates other than by a `q` command (by hangup, deleting its window, etc.), modified files are saved in an executable file, `$home/sam.save`. This program, when executed, asks whether to write each file back to a external file. The answer `y` causes writing; anything else skips the file.

FILES

`$home/sam.save`
`$home/sam.err`
`/sys/lib/samsave` the program called to unpack `$home/sam.save`.

SEE ALSO

ed(1), *sed*(1), *grep*(1), *8½*(1), *regexp*(6).

NAME

sed − stream editor

SYNOPSIS

`sed` [`-n`] [`-e` *script*] [`-f` *sfile*] [*file* ...]

DESCRIPTION

Sed copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The `-f` option causes the script to be taken from file *sfile*; these options accumulate. If there is just one `-e` option and no `-f`'s, the flag `-e` may be omitted. The `-n` option suppresses the default output.

A script consists of editing commands, one per line, of the following form:

> [*address* [`,` *address*]] *function* [*argument* ...]

In normal operation *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a `D` command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under `-n`) and deletes the pattern space.

An *address* is either a decimal number that counts input lines cumulatively across files, a `$` that addresses the last line of input, or a context address, `/`*regular-expression*`/`, in the style of *regexp*(6), with the added convention that `\n` matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied to non-selected pattern spaces by use of the negation function `!` (below).

An argument denoted *text* consists of one or more lines, all but the last of which end with `\` to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an `s` command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted *rfile* or *wfile* must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 120 distinct *wfile* arguments.

`a\` *text*	Append. Place *text* on the output before reading the next input line.
`b` *label*	Branch to the **:** command bearing the *label.* If *label* is empty, branch to the end of the script.
`c\` *text*	Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.
`d`	Delete the pattern space. Start the next cycle.
`D`	Delete the initial segment of the pattern space through the first newline. Start the next cycle.
`g`	Replace the contents of the pattern space by the contents of the hold space.
`G`	Append the contents of the hold space to the pattern space.
`h`	Replace the contents of the hold space by the contents of the pattern space.

`H`	Append the contents of the pattern space to the hold space.
`i\` *text*	Insert. Place *text* on the standard output.
`n`	Copy the pattern space to the standard output. Replace the pattern space with the next line of input.
`N`	Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)
`p`	Print. Copy the pattern space to the standard output.
`P`	Copy the initial segment of the pattern space through the first newline to the standard output.
`q`	Quit. Branch to the end of the script. Do not start a new cycle.
`r` *rfile*	Read the contents of *rfile*. Place them on the output before reading the next input line.
`s/`*regular-expression*`/`*replacement*`/`*flags*	Substitute the *replacement* string for instances of the *regular-expression* in the pattern space. Any character may be used instead of `/`. For a fuller description see *regexp*(6). *Flags* is zero or more of `g` Global. Substitute for all non-overlapping instances of the *regular expression* rather than just the first one. `p` Print the pattern space if a replacement was made. `w` *wfile* Write. Append the pattern space to *wfile* if a replacement was made.
`t` *label*	Test. Branch to the `:` command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a `t`. If *label* is empty, branch to the end of the script.
`w`	*wfile* Write. Append the pattern space to *wfile.*
`x`	Exchange the contents of the pattern and hold spaces.
`y/`*string1*`/`*string2*`/`	Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2.* The lengths of *string1* and *string2* must be equal.
`!`*function*	Don't. Apply the *function* (or group, if *function* is `{}`) only to lines *not* selected by the address(es).
`:` *label*	This command does nothing; it bears a *label* for `b` and `t` commands to branch to.
`=`	Place the current line number on the standard output as a line.
`{`	Execute the following commands through a matching `}` only when the pattern space is selected.

An empty command is ignored.

EXAMPLES

`sed 10q file`

Print the first 10 lines of the file.

`sed '/^$/d'`

Delete empty lines from standard input.

`sed 's/UNIX/& system/g'`

Replace every instance of `UNIX` by `UNIX system`.

```
sed 's/ *$//          drop trailing blanks
/^$/d                 drop empty lines
s/  */\               replace blanks by newlines
/g
/^$/d' chapter*
```

Print the files `chapter1`, `chapter2`, etc. one word to a line.

```
nroff -ms manuscript | sed '
${
      /^$/p           if last line of file is empty, print it
}
//N                   if current line is empty, append next line
/^\n$/D'              if two lines are empty, delete the first
```

Delete all but one of each group of empty lines from a formatted manuscript.

SEE ALSO

ed(1), *grep*(1), *awk*(1), *lex*(1), *sam*(1), *regexp*(6)

L. E. McMahon, 'SED — A Non-interactive Text Editor', Unix Research System Programmer's Manual, Volume 2.

BUGS

If input is from a pipe, buffering may consume characters beyond a line on which a `q` command is executed.

NAME

seq − print sequences of numbers

SYNOPSIS

`seq` [`-w`] [`-f`*format*] [*first* [*incr*]] *last*

DESCRIPTION

Seq prints a sequence of numbers, one per line, from *first* (default 1) to as near *last* as possible, in increments of *incr* (default 1). The numbers are interpreted as floating point.

Normally integer values are printed as decimal integers. The options are

`-f`*format*	Use the *print*(2)-style *format print* for printing each (floating point) number. The default is `%g`.
`-w`	Equalize the widths of all numbers by padding with leading zeros as necessary. Not effective with option `-f`, nor with numbers in exponential notation.

EXAMPLES

`seq 0 .05 .1`

Print `0 0.05 0.1` (on separate lines).

`seq -w 0 .05 .1`

Print `0.00 0.05 0.10`.

BUGS

Option `-w` always surveys every value in advance. Thus `seq -w 1000000000` is a hopeless way to get an ‘infinite’ sequence.

NAME

size − print size of executable files

SYNOPSIS

`size` [*file* ...]

DESCRIPTION

Size prints the size of the segments for each of the argument executable files (default `v.out`). The format is

> *textsize*`t` + *datasize*`d` + *bsssize*`b` = *total*

where the numbers are in bytes.

NAME

sleep − suspend execution for an interval

SYNOPSIS

`sleep` *time*

DESCRIPTION

Sleep suspends execution for *time* seconds.

EXAMPLES

Execute a command 100 seconds hence.

```
{sleep 100; command}&
```

Repeat a command every 30 seconds.

```
while (~ 1 1){
      command
      sleep 30
}
```

SEE ALSO

sleep(2)

NAME

sort − sort and/or merge files

SYNOPSIS

`sort` [`-cmuMbdfinrt`*x*] [+*pos1* [−*pos2*] ...] ... [`-k` *pos1* [,*pos2*]] ... [`-o` *output*] [*option* ...] [*file* ...]

DESCRIPTION

Sort sorts lines of all the *files* together and writes the result on the standard output. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by runes. The ordering is affected globally by the following options, one or more of which may appear.

`-M`	Compare as months. The first three non-white space characters of the field are folded to upper case and compared so that `JAN` precedes `FEB`, etc. Invalid fields compare low to `JAN`.
`-b`	Ignore leading white space (spaces and tabs) in field comparisons.
`-d`	‘Phone directory’ order: only letters, accented letters, digits and white space are significant in comparisons.
`-f`	Fold lower case letters onto upper case. Accented characters are folded to their non-accented upper case form.
`-i`	Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.
`-n`	An initial numeric string, consisting of optional white space, optional plus or minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value.
`-g`	Numbers, like `-n` but with optional `e`-style exponents, are sorted by value.
`-r`	Reverse the sense of comparisons.
`-t`*x*	‘Tab character’ separating fields is *x*.

The notation +*pos1* −*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m*`.`*n*`,` optionally followed by one or more of the flags `Mbdfginr`, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. A missing `.`*n* means `.0;` a missing −*pos2* means the end of the line. Under the `-t`*x* option, fields are strings separated by *x*; otherwise fields are non-empty strings separated by white space. White space before a field is part of the field, except under option `-b`. A `b` flag may be attached independently to *pos1* and *pos2*.

The notation `-k` *pos1 [, pos2]* is the Posix version of fields. Of course, *pos1* and *pos2* have the same format but different meanings. The value of `.`*m* is base 1 instead of base 0 and a missing `.`*n* in *pos2* is the end of the field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

`-c`	Check that the single input file is sorted according to the ordering rules; give no output unless the file is out of sort.
`-m`	Merge; the input files are already sorted. `-r` option. This option does not work with the `-u` option.
`-u`	Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.
`-o`	The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

`−T`*dir* Put temporary files in *dir* rather than in `/tmp`.

EXAMPLES

```
sort −u +0f +0 list
```

Print in alphabetical order all the unique spellings in a list of words where capitalized words differ from uncapitalized.

```
sort −t: +1 /adm/users
```

Print the users file sorted by user name (the second colon-separated field).

```
sort −umM dates
```

Print the first instance of each month in an already sorted file. Options `−um` with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
grep −n '^' input | sort −t: +1f +0n | sed 's/[0−9]*://'
```

A stable sort: input lines that compare equal will come out in their original order.

FILES

`/tmp/sort.`<*pid*>.<*ordinal*>

SEE ALSO

uniq(1), *look*(1)

DIAGNOSTICS

Sort comments and exits with non-null status for various trouble conditions and for disorder discovered under option `−c`.

BUGS

An external null character can be confused with an internally generated end-of-field character. The result can make a sub-field not sort less than a longer field.

Some of the options, e.g. `−i` and `−M`, are hopelessly provincial.

NAME

spell − find spelling errors

SYNOPSIS

`spell` [*options*] (`.,.`)[*file*] (`.,.`).PP `sprog` [*options*] [`-f` *file*]

DESCRIPTION

Spell looks up words from the named *files* (standard input default) in a public spelling list. Possible misspellings—words that occur in neither and are not plausibly derivable from the former—are placed on the standard output.

Spell ignores constructs of *troff*(1) and its standard preprocessors. It understands these options:

`-b`	Check British spelling.
`-v`	Print all words not literally in the spelling list, with derivations.
`-x`	Print, marked with =, every stem as it is looked up in the spelling list, along with its affix classes.

As a matter of policy, *spell* does not admit multiple spellings of the same word. Variants that follow general rules are preferred over those that don't, even when the unruly spelling is more common. Thus, in American usage, 'modelled', 'sizeable', and 'judgment' are rejected in favor of 'modeled', 'sizable', and 'judgement'. Agglutinated variants are shunned: 'crewmember' and 'backyard' cede to 'crew member' and 'back yard' (noun) or 'back-yard' (adjective).

FILES

`/sys/lib/amspell`	American spelling list
`/sys/lib/brspell`	British spelling list
`/bin/aux/sprog`	The actual spelling checker. It expects one word per line on standard input, and takes the same arguments as *spell*.

SEE ALSO

deroff(1)

BUGS

The heuristics of *deroff*(1) used to excise formatting information are imperfect.

The spelling list's coverage is uneven; in particular biology, medicine, and chemistry, and perforce proper names, not to mention languages other than English, are covered very lightly.

NAME

split − split a file into pieces

SYNOPSIS

`split` [*option* ...] [*file*]

DESCRIPTION

Split reads *file* (standard input by default) and writes it in pieces of 1000 lines per output file. The names of the output files are `xaa`, `xab`, and so on to `xzz`. The options are

−*n* Split into *n*-line pieces.

`-e` *expression*
File divisions occur at each line that matches a regular *expression*; see *regexp*(6). Multiple `-e` options may appear. If a subexpression of *expression* is contained in parentheses `(...)`, the output file name is the portion of the line which matches the subexpression.

`-f` *stem*
Use *stem* instead of `x` in output file names.

`-s` *suffix*
Append *suffix* to names identified under `-e`.

`-x` Exclude the matched input line from the output file.

`-i` Ignore case in option `-e`; force output file names (excluding the suffix) to lower case.

SEE ALSO

sed(1), *awk*(1) *grep*(1)

NAME

stop, start − print commands to stop and start processes

SYNOPSIS

`stop` *name*

`start` *name*

DESCRIPTION

Stop prints commands that will cause all processes called *name* and owned by the current user to be stopped. The processes can then be debugged when they are in a consistent state.

Start prints commands that will cause all stopped processes called *name* and owned by the current user to be started again.

Use the `send` command of *8½*(1), or pipe into *rc*(1) to execute the commands.

SEE ALSO

ps(1), *kill*(1), *proc*(3)

NAME

strings − extract printable strings

SYNOPSIS

`strings` [*file* ...]

DESCRIPTION

Strings finds and prints strings containing 6 consecutive printable runes in a (typically) binary file. If the *file* argument is omitted input is taken from standard input. Strings reports the decimal offset within the file at which the string starts and the text of the string. If the string is longer than 70 runes the line is terminated by three dots and the printing is resumed on the next line with the offset of the continuation line.

SEE ALSO

nm(1)

NAME

strip − remove symbols from binary files

SYNOPSIS

`strip` *file* ...

DESCRIPTION

Strip removes symbol table segments from executable files. Stripping a file requires write permission of the file and the directory it is in.

SEE ALSO

a.out(6)

NAME

sum − sum and count blocks in a file

SYNOPSIS

`sum` [`-5r`] [*file* ...]

DESCRIPTION

By default, *sum* calculates and prints a 32-bit hexadecimal checksum, a byte count and the name of each *file*. The checksum is also a function of the input length. If no files are given, the standard input is summed. Other summing algorithms are available. The options are

`-r` Sum with the algorithm of System V's `sum -r` and print the length (in 1K blocks) of the input.

`-5` Sum with System V's default algorithm and print the length (in 512-byte blocks) of the input.

Sum is typically used to look for bad spots, to validate a file communicated over some transmission line or as a quick way to determine if two files on different machines might be the same.

SEE ALSO

cmp(1), *wc*(1)

NAME

syscall − test a system call

SYNOPSIS

`syscall` [`-o`] *entry* [*arg* ...]

DESCRIPTION

Syscall makes the *entry* system call with the given arguments. It prints the return value and the error string, if there was an error. An argument is either an integer constant as in C (its value is passed), a string (its address is passed), or the literal `buf` (a pointer to a 1 Kbyte buffer is passed). If `-o` is given, the contents of the 1 Kbyte buffer are printed as a string after the system call is done.

EXAMPLES

Write a string to standard output

```
syscall write 1 hello 5
```

Print the last system call error string

```
syscall -o errstr buf
```

SEE ALSO

Section 2 of this manual.

DIAGNOSTICS

If *entry* is not a system call name, the exit status is `unknown`. If the system call succeeds, the exit status is null; otherwise the exit status is the string that *errstr*(2) returns.

NAME

tail − deliver the last part of a file

SYNOPSIS

`tail` [+−*number*[`lbc`][`rf`]] [*file*]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is copied.

Copying begins at position +*number* measured from the beginning, or −*number* from the end of the input. *Number* is counted in lines, 1K blocks or characters, according to the appended flag `l`, `b`, or `c`. Default is `−10l` (ten ell).

The further flag `r` causes tail to print lines from the end of the file in reverse order; `f` (follow) causes *tail,* after printing to the end, to keep watch and print further data as it appears.

EXAMPLES

`tail file`

Print the last 10 lines of a file.

`tail +0f file`

Print a file, and continue to watch data accumulate as it grows.

`sed 10q file`

Print the first 10 lines of a file.

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length.

According to custom, option +*number* counts lines from 1, and counts blocks and characters from 0.

NAME

tapefs – mount archival file systems

SYNOPSIS

`fs/32vfs` [`-m` *mountpoint*] [`-p` *passwd*] [`-g` *group*] *file*
`fs/cpiofs`
`fs/tarfs`
`fs/tpfs`
`fs/v6fs`

DESCRIPTION

These commands interpret data from traditional tape or file system formats stored in *file*, and mount their contents (read-only) into a Plan 9 file system. The optional `-p` and `-g` flags specify Unix-format password (respectively group) files that give the mapping between the numeric user- and group-ID numbers on the media and the strings reported by Plan 9 status inquiries. The `-m` flag introduces the name at which the new file system should be attached.

32vfs interprets raw disk dumps of 32V systems, which are ca. 1978 research Unix systems for the VAX, and also pre-FFS Berkeley VAX systems.

Cpiofs interprets *cpio* tape images (constructed with `c` flag).

Tarfs interprets *tar* tape images.

Tpfs interprets *tp* tapes from the Fifth through Seventh Edition research Unix systems.

V6fs interprets disk images from the Fifth and Sixth edition research Unix systems.

These commands are constructed in a highly stereotyped way using the files *fs.c* and *util.c* in their source directory, which in turn derive substantially from *ramfs*(4).

SEE ALSO

Section 5 *passim*, *ramfs*(4).

NAME

tar − archiver

SYNOPSIS

`tar` *key* [*file* ...]

DESCRIPTION

Tar saves and restores file trees. It is most often used to transport a tree of files from one system to another. The *key* is a string that contains at most one function letter plus optional modifiers. Other arguments to the command are names of files or directories to be dumped or restored. A directory name implies all the contained files and subdirectories (recursively).

The function is one of the following letters:

`c`	Create a new archive with the given files as contents.
`x`	Extract the named files from the archive. If a file is a directory, the directory is extracted recursively. Modes are restored if possible. If no file argument is given, extract the entire archive. If the archive contains multiple entries for a file, the latest one wins.
`t`	List all occurrences of each *file* in the archive, or of all files if there are no *file* arguments.
`r`	The named files are appended to the archive.

The modifiers are:

`v`	(verbose) Print the name of each file treated preceded by the function letter. With `t`, give more details about the archive entries.
`f`	Use the next argument as the name of the archive instead of the default standard input (for keys `x` and `t`) or standard output (for keys `c` and `r`).

EXAMPLES

Tar can be used to move hierarchies thus:

```
{cd fromdir; tar c .} | {cd todir; tar x}
```

SEE ALSO

ar(1), *bundle*(1)

BUGS

There is no way to ask for any but the last occurrence of a file.
File path names are limited to 100 characters.
The tar format allows specification of links and symbolic links, concepts foreign to Plan 9: they are ignored.

NAME

tbl − format tables for nroff or troff

SYNOPSIS

`tbl` [*file* ...]

DESCRIPTION

Tbl is a preprocessor for formatting tables for *nroff* or *troff*(1). The input files are copied to the standard output, except for segments of the form

```
.TS
options ;
format .
data
.T&
format .
data
...
.TE
```

which describe tables and are replaced by *troff* requests to lay out the tables. If no arguments are given, *tbl* reads the standard input.

The (optional) *options* line is terminated by a semicolon and contains one or more of

`center`	center the table; default is left-adjust
`expand`	make table as wide as current line length
`box`	
`doublebox`	enclose the table in a box or double box
`allbox`	enclose every item in a box
`tab(`*x*`)`	use *x* to separate input items; default is tab
`linesize(`*n*`)`	set rules in *n*-point type
`delim(`*xy*`)`	recognize *x* and *y* as *eqn*(1) delimiters

Each line, except the last, of the obligatory *format* describes one row of the table. The last line describes all rows until the next `.T&`, where the format changes, or the end of the table at `.TE`. A format is specified by key letters, one per column, upper or lower case

`L`	Left justify: the default for columns without format keys.
`R`	Right justify.
`C`	Center.
`N`	Numeric: align at decimal point (inferred for integers) or at `\&`.
`S`	Span: extend previous column across this one.
`A`	Alphabetic: left-aligned within column, widest item centered, indented relative to `L` rows.
`^`	Vertical span: continue item from previous row into this row.
`-`	Draw a horizontal rule in this column.
`_`	Draw a double horizontal rule in this column.

Key letters may be followed by modifiers, also either case:

|	Draw vertical rule between columns.
||	Draw a double vertical rule between columns.
n	Gap between column is *n* ens wide. Default is 3.
`F`*font*	Use specified *font.* `B` and `I` mean `FB` and `FI`.
`T`	Begin vertically-spanned item at top row of range; default is vertical centering (with `^`).
`P`*n*	Use point size *n.*
`V`*n*	Use *n*-point vertical spacing in text block; signed *n* means relative change.
`W(`*n*`)`	Column width as a *troff* width specification. Parens are optional if *n* is a simple integer.

`E`	Equalize the widths of all columns marked `E`.

Each line of *data* becomes one row of the table; tabs separate items. Lines beginning with `.` are *troff* requests. Certain special data items are recognized:

`_`	Draw a horizontal rule in this column.
`=`	Draw a double horizontal rule in this column. A data line consisting of a single _ or = draws the rule across the whole table.
`\_`	Draw a rule only as wide as the contents of the column.
`\R`*x*	Repeat character *x* across the column.
`\^`	Span the previous item in this column down into this row.
`T{`	The item is a text block to be separately formatted by *troff* and placed in the table. The block continues to the next line beginning with `T}`. The remainder of the data line follows at that point.

When it is used in a pipeline with *eqn,* the *tbl* command should be first, to minimize the volume of data passed through pipes.

EXAMPLES

Let <tab> represent a tab (which should be typed as a genuine tab).

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town<tab>Households
<tab>Number<tab>Size
Bedminster<tab>789<tab>3.26
Bernards Twp.<tab>3087<tab>3.74
Bernardsville<tab>2018<tab>3.30
.TE
```

```
         Household Population
          Town        Households
                      Number  Size
Bedminster              789   3.26
Bernards Twp.          3087   3.74
Bernardsville          2018   3.30
```

SEE ALSO

troff(1), *eqn*(1)

M. E. Lesk and L. L. Cherry, 'TBL—a Program to Format Tables', Unix Research System Programmer's Manual, Volume 2

NAME

tcs − translate character sets

SYNOPSIS

`tcs` [`-slcv`] [`-f` *ics*] [`-t` *ocs*] [*file* ...]

DESCRIPTION

Tcs interprets the named *file(s)* (standard input default) as a stream of characters from the *ics* character set or format, converts them to runes, and then converts them into a stream of characters from the *ocs* character set or format on the standard output. The default value for *ics* and *ocs* is `utf`, the UTF encoding described in *utf*(6). The `-l` option lists the character sets known to *tcs*. Processing continues in the face of conversion errors (the `-s` option prevents reporting of these errors). The `-c` option forces the output to contain only correctly converted characters; otherwise, `0x80` characters will be substituted for UTF encoding errors and `0xFFFD` characters will substituted for unknown characters.

The `-v` option generates various diagnostic and summary information on standard error, or makes the `-l` output more verbose.

Tcs recognizes the following character sets:

`utf`	(our) UTF encoding from X/Open
`utf1`	UTF encoding from 10646
`ascii`	7-bit ASCII
`8859-1`	Latin-1 (Central European)
`8859-2`	Latin-2 (Czech .. Slovak)
`8859-3`	Latin-3 (Dutch .. Turkish)
`8859-4`	Latin-4 (Scandinavian)
`8859-5`	Part 5 (Cyrillic)
`8859-6`	Part 6 (Arabic)
`8859-7`	Part 7 (Greek)
`8859-8`	Part 8 (Hebrew)
`8859-9`	Latin-5 (Finnish .. Portuguese)
`kio8`	KIO-8 (recommended)
`kio8x`	KIO-8 (alternate)
`jis`	XJIS
`gb`	Chinese national standard (GB2312-80)
`big5`	Big 5 (HKU version)
`unicode`	Unicode 1.0

EXAMPLES

```
tcs -f 8859-1
```

Convert 8859-1 (Latin-1) characters into UTF format.

```
tcs -s -f jis
```

Convert characters encoded in one of several shift JIS encodings into UTF format. Unknown Kanji will be converted into `0xFFFD` characters.

```
tcs -lv
```

Print an up to date list of the supported character sets.

SEE ALSO

ascii(1), *rune*(2), *utf*(6).

NAME

tee − pipe fitting

SYNOPSIS

`tee` [`-i`] [`-a`] [`-u`] [*file* ...]

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the *files.* The options are

`-i`	Ignore interrupts.
`-a`	Append the output to the *files* rather than rewriting them.
`-u`	Unbuffered: write the output one character at a time.

NAME

tel, pq − look in phone book

SYNOPSIS

`tel` *key* ...

`pq` *name* ...

DESCRIPTION

Tel looks up *key* in a private telephone book, `$home/lib/tel,` and in the public telephone book, `/lib/tel`. It uses *grep* (with the `-i` option to ignore case differences), so the key may be any part of a name or number. Customarily, the telephone book contains names, userids, home numbers, and office numbers of users. It also contains a directory of area codes and miscellaneous people of general interest.

Pq looks up names in the AT&T personnel database. *Name* should be a surname optionally prefixed by initials and periods, as in `emlin` or `g.emlin` or `g.r.emlin`. *Pq* also accepts keyword arguments of the form *key=value. Key* may be one of `org` for organization number, `ext` or `tel` for office telephone extension, or `room` for office number. Beware that = must be quoted when passed through *rc*(1).

FILES

`/lib/tel`	Public telephone number database.

NAME

test − set status according to condition

SYNOPSIS

`test` *expr*

DESCRIPTION

Test evaluates the expression *expr*. If the value is true the exit status is null; otherwise the exit status is non-null. If there are no arguments the exit status is non-null.

The following primitives are used to construct *expr*.

`-r` *file*	True if the file exists (is accessible) and is readable.
`-w` *file*	True if the file exists and is writable.
`-x` *file*	True if the file exists and has execute permission.
`-e` *file*	True if the file exists.
`-f` *file*	True if the file exists and is a plain file.
`-d` *file*	True if the file exists and is a directory.
`-s` *file*	True if the file exists and has a size greater than zero.
`-t` *fildes*	True if the open file whose file descriptor number is *fildes* (1 by default) is the same file as `/dev/cons`.
s1 `=` *s2*	True if the strings *s1* and *s2* are identical.
s1 `!=` *s2*	True if the strings *s1* and *s2* are not identical.
s1	True if *s1* is not the null string. (Deprecated.)
`-z` *s1*	True if the length of string *s1* is zero.
n1 `-eq` *n2*	True if the integers *n1* and *n2* are arithmetically equal. Any of the comparisons `-ne`, `-gt`, `-ge`, `-lt`, or `-le` may be used in place of `-eq`. The (nonstandard) construct `-l` *string,* meaning the length of *string,* may be used in place of an integer.

These primaries may be combined with the following operators:

`!`	unary negation operator
`-o`	binary *or* operator
`-a`	binary *and* operator; higher precedence than `-o`
`(` *expr* `)`	parentheses for grouping.

Notice that all the operators and flags are separate arguments to *test*. Notice also that parentheses and equal signs are meaningful to *rc* and must be enclosed in quotes.

EXAMPLES

Test is a dubious way to check for specific character strings: it uses a process to do what an *rc*(1) match or switch statement can do. The first example is not only inefficient but wrong, because *test* understands the purported string `"-c"` as an option.

```
if (test $1 '=' "-c") echo OK # wrong!
```

A better way is

```
if (~ $1 -c) echo OK
```

Test whether `abc` is in the current directory.

```
test -f abc -o -d abc
```

SEE ALSO

rc(1)

NAME

tex, latex, slitex, bibtex, dvips, dviselect, mf − text formatting and typesetting

SYNOPSIS

`tex` [*first-line*]

`latex` *file*[.tex]

`slitex` *file*[.tex]

`dvips` [*option ...*] *dvifile*

`dviselect` [`−s`] [`−i` *infile*] [`−o` *outfile*] *list of pages* [*infile* [*outfile*]]

`bibtex` *auxname*

`mf` [*first-line*]

DESCRIPTION

Tex formats interspersed text and commands and outputs a `.dvi` ('device independent') file.

An argument given on the command line behaves as the first input line. That line should begin with a (possibly truncated) file name or a *controlsequence*. Thus `tex paper` processes the file `paper.tex`. The base name of `paper` becomes the *jobname*, and is used in forming output file names. If no file is named, the jobname is `texput`. The default `.tex` extension can be overridden by specifying an extension explicitly.

The output is written on *jobname*`.dvi`, which can be printed using *lp*(1). A log of error messages goes into *jobname*`.log`.

As well as the standard TeX fonts, many Postscript fonts can be used (see the contents of `/sys/lib/tex/fonts/psvf`). The file `testfont.tex` (in the standard macro directory) will print a table of any font.

These environment variables adjust the behavior of *tex*:

`TEXINPUTS`	Search path for `\input` and `\openin` files. It should be colon-separated, and start with dot. Default: `.:/sys/lib/tex/macros`
`TEXFONTS`	Search path for font metric files. Default: `/sys/lib/tex/fonts/tfm`
`TEXFORMATS`	Search path for format files. Default: `/sys/lib/tex/macros`
`TEXPOOL`	Search path for strings. Default: `/sys/lib/tex`
`TEXEDIT`	Template for the switch-to-editor-on-error option, with `%s` for the file name and `%d` for the line number. Default: `/bin/ed %s`

Latex is a version of *tex* with a standard set of macros loaded. *Latex* produces *file*`.dvi` and a cross-referencing file, *file*`.aux`. It might be necessary to run *latex* twice, to get all of the cross-referencing done properly. *Slitex* is a variant of *latex* with fonts and commands suitable for making slides.

Bibtex is a bibliography processing program, often used in conjunction with *latex*. *Bibtex* reads the top-level auxiliary (`.aux`) file output by *latex* and creates a bibliography (`.bbl`) file to be included in the LaTeX source file. The *auxname* on the command line should be given without an extension. Each `\cite` in the source file is looked up in bibliography files to gather together those used in the document. Then a bibliography style file is executed to write a `\thebibliography` environment.

The source file should have defined the bibliography (`.bib`) files to search with the `\bibliography` command, and the bibliography style (`.bst`) file to execute with the `\bibliographystyle` command. *Bibtex* searches the `TEXINPUTS` path for `.bst` files, and the `BIBINPUTS` path for `.bst` files. The LaTeX manual describes how to make bibliography files.

Dvips converts `.dvi` files to Postscript, writing the result on standard output. It is normally invoked by *lp*(1), but if invoked separately, the following options are useful:

`−r` reverse pages. `−r0` means don't reverse pages (if reversing is default).

`-T`*dev*	output device: *dev* is one of `laserwriter` (default for *dvips*), `gnot`, `fax`, or `lino` (the computer center's high resolution Postscript service). The `-Tgnot` option should be used for preparing output for *psi*(1).
`-L`	print paper in landscape mode.
`-Z`	compress the fonts before sending them.
`-Z0`	don't compress the fonts before sending them.

The following environment variables affect *dvips*:

`TEXPKS`	Search path for font bitmaps (PK files).
`TEXVFONTS`	Search path for virtual font descriptions.

Dviselect selects pages from a `.dvi` file, creating a new `.dvi` file. A *range* is a string of the form *first:last* where both *first* and *last* are optional numeric strings, with negative numbers indicated by a leading underscore character (_). If both *first* and *last* are omitted, the colon may also be omitted, or may be replaced with an asterisk (`*`). A *TeX page selector* is a list of pages separated by periods. A *list of pages* is described by a set of page TeX page selectors, separated by commas and/or white space. *Dviselect* actually looks at the ten *count* variables that TeX writes; the first of these (`\count0`) is the page number, with `\count1` through `\count9` having varied uses depending on which macro packages are in use. (Typically `\count1` is a chapter or section number.) A page is included in *dviselect*'s output if all its \count values are within any one of the ranges listed on the command line. For example, the command `dviselect *.1,35:` might select everything in chapter 1, as well as pages 35 and up.

Instead of `\count` values, *dviselect* can also select by absolute page number, indicated by a leading equal sign (=). Ranges of absolute pages are also allowed: `dviselect =3:7` will extract the third through seventh pages.

Dvips understands some extended graphics commands that can be output using *tpic specials* in the TeX source. Many of them work by building up a path of *x,y* pairs, and then doing something with the path. The tpic coordinate system has its origin at the current dvi position when a drawing special is emitted; all length arguments are in units of milli-inches, and the y-axis goes positive downward.

`\special{pa` *x y*`}`
Add *x,y* to the current path.

`\special{fp}`
Flush the current path: draw it as a polygonal line and reset the path to be empty.

`\special{da` *dlen*`}`
Like `fp` but draw dashed line, with dashes *dlen* milli-inches long.

`\special{dt` *slen*`}`
Like `fp` but draw a dotted line, with dots *slen* apart.

`\special{sp}`
Like `fp` but draw a quadratic spline. The spline goes through the midpoints of the segments of the path, and straight pieces extend it to the endpoints.

`\special{ar` *x y xr yr s e*`}`
Draw a circular or elliptical arc with center at *x,y* and radii *xr* and *yr*. The arc goes clockwise from angle *s* to angle *e* (angles measured clockwise from the positive x-axis).

`\special{pn` *n*`}`
Set line width (pen diameter) to *n* milli-inches.

`\special{bk}`
Set shading to black (will fill the next object drawn with black).

`\special{sh}`
Set shading to grey.

`\special{wh}`
Set shading to white.

`\special{psfile=`*file options*`}`
Include *file*, which should be a Postscript illustration, making its origin be the current dvi position. The default Postscript transformation matrix will be in effect, but it can be modified by the *options*, a list of space-separated *key*=*value* assignments. Allowed keys are: hoffset, voffset, hscale, vscale, angle, If supplied, these values are supplied to Postscript *translate*,*scale*, and *rotate* commands, in that order. Also, keys hsize and vsize may be supplied, to cause clipping to those sizes. Sizes and offsets should be specified in points, angles should be specified in degrees.

All of the specials leave TeX at the same position on the page that it started in.

Mf runs metafont, program that produces fonts for TeX. It is used by *dvips* when bitmaps for a given font at a given size do not exist.

FILES

`/sys/lib/tex/macros/*`	macros and preloaded format files
`/sys/lib/tex/macros/doc/*`	more TeX-related documentation
`/sys/lib/tex/fonts/tfm`	font metrics
`/sys/lib/tex/fonts/psvf`	PostScript virtual font metrics
`/sys/lib/tex/fonts/canonpk`	bitmaps for canon engines (300 dpi)
`/sys/lib/tex/fonts/linopk`	bitmaps for Linotron (1270 dpi)
`/sys/lib/tex/fonts/gnotpk`	bitmaps for gnot screen (100 dpi)
`/sys/lib/tex/*`	miscellaneous configuration files and Postscript headers

SEE ALSO

pic(1), *lp*(1), *proof*(1), *psi*(1), *troff*(1
D. E. Knuth, *The TEXbook,* Addison-Wesley, 1984
L. Lamport, *LaTeX, A Document Preparation System,* Addison-Wesley, 1985
H. Trickey, ‘Latex User Guide’, Unix Research System Programmer’s Manual, Volume 2
Various documents in `/sys/lib/tex/macros/doc.`

BUGS

Should be spelled τεχ.

NAME

think – HP ThinkJet filter

SYNOPSIS

`think` [`-r`] [`-o` *outfile*] [*file* ...]

DESCRIPTION

Think filters the given *files* (standard input by default) to *outfile* (`/dev/lpt1data` by default). Tabs are expanded, carriage-returns are added after newlines, and runes that fall outside the standard ASCII range are converted to Roman-8 codes if possible. The `-r` option suppresses this conversion.

FILES

`/dev/lpt1data`	Centronix printer port

NAME

time − time a command

SYNOPSIS

`time` *command* [*arg* ...]

DESCRIPTION

The *command* is executed with the given arguments; after it is complete, *time* reports on standard error the program’s elapsed user time, system time, and real time, in seconds, followed by the command line.

SEE ALSO

prof(1)

NAME

touch − set modification date of a file

SYNOPSIS

`touch` [`-c`] *file* ...

DESCRIPTION

Touch attempts to set the modification time of the *files* to the current time. If a *file* does not exist, it will be created unless option `-c` is present.

SEE ALSO

ls(1), *stat*(2), *chmod*(1)

BUGS

Touch will not touch directories.

NAME

tr − translate characters

SYNOPSIS

`tr` [`-cds`] [*string1* [*string2*]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. When *string2* is short it is padded to the length of *string1* by duplicating its last character. Any combination of the options `-cds` may be used:

`-c` Complement *string1*: replace it with a lexicographically ordered list of all other 8-bit unsigned characters.

`-d` Delete from input all characters in *string1*.

`-s` Squeeze repeated output characters that occur in *string2* to single characters.

In either string a noninitial sequence *−x,* where *x* is any character (possibly quoted), stands for a range of characters: a possibly empty sequence of codes running from the successor of the previous code up through the code for *x.* The character \ followed by 1, 2 or 3 octal digits stands for the character whose Unicode value is given by those digits. The character sequence `\x` followed by 1, 2, 3, or 4 hexadecimal digits stands for the character whose Unicode value is given by those digits. A \ followed by any other character stands for that character.

EXAMPLES

Replace all upper-case letters by lower-case.

```
tr A-Z a-z <mixed >lower
```

Create a list of all the words in `file1` one per line in `file2`, where a word is taken to be a maximal string of alphabetics. *String2* is given as a quoted newline.

```
tr -cs A-Za-z '
' <file1 >file2
```

SEE ALSO

sed(1)

NAME

troff, nroff − text formatting and typesetting

SYNOPSIS

`troff` [*option ...*] [*file ...*]

`nroff` [*option ...*] [*file ...*]

DESCRIPTION

Troff formats text in the named *files* for printing on a typesetter. *Nroff* does the same, but produces output suitable for typewriter-like devices.

If no *file* argument is present, the standard input is read. An argument consisting of a single minus (`-`) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

`-o`*list*	Print pages in the comma-separated *list* of numbers and ranges. A range *N*−*M* means *N* through *M*; initial −*M* means up to *M;* final *N*− means from *N* to the end.
`-n`*N*	Number first generated page *N*.
`-m`*name*	Process the macro file `/sys/lib/tmac/tmac.`*name* before the input *files*.
`-r`*aN*	Set register *a* (one character name) to *N*.
`-i`	Read standard input after the input files are exhausted.
`-q`	Invoke the simultaneous input-output mode of the `rd` request.
`-N`	Produce output suitable for typewriter-like devices.

Typesetter devices (not `-N`) only

`-a`	Send a printable textual approximation of the results to the standard output.
`-T`*dest*	Prepare output for typesetter *dest*: `-TLatin1` (The default.) PostScript printers with ISO 8859-1 Latin-1 and other characters. The name is a holdover; in fact the output is Unicode characters encoded in UTF. `-Tpost` PostScript printers `-T202` Mergenthaler Linotron 202
`-F`*dir*	Take font information from directory *dir.*

Typewriter (`-N`) output only

`-s`*N*	Halt prior to every *N* pages (default *N*=1) to allow paper loading or changing.
`-T`*name*	Prepare output for specified terminal. Known *names* include `utf` for the normal Plan 9 Unicode/UTF character set (default), `37` for the Teletype model 37, `lp` (‘line-printer’) for any terminal without half-line capability, `450` for the DASI-450 (Diablo Hyterm), and `think` (HP ThinkJet).
`-e`	Produce equally-spaced words in adjusted lines, using full terminal resolution.
`-h`	Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

FILES

`/tmp/trtmp*`	temporary file
`/sys/lib/tmac/tmac.*`	standard macro files
`/sys/lib/troff/term/*`	terminal driving tables for *nroff*
`/sys/lib/troff/font/*`	font width tables for *troff*

SEE ALSO

lp(1), *proof*(1), *eqn*(1), *tbl*(1), *pic*(1), *grap*(1), *doctype*(1), *ms*(6), *mpm*(6), *bitmap*(6), *tex*(1)

J. F. Ossanna and B. W. Kernighan, ‘Nroff/Troff User’s Manual’, Unix Research System Programmer’s Manual, Volume 2

B. W. Kernighan, ‘A TROFF Tutorial’, *ibid.*

NAME

tweak − edit bitmap files, subfont files, face files, etc.

SYNOPSIS

`tweak` [*file* ...]

DESCRIPTION

Tweak edits existing files holding various forms of bitmap images. To create original images, use *art*(1).

Tweak reads its argument *files* and displays the resulting bitmaps in a vertical column. If the bitmap is too wide to fit across the display, it is folded much like a long line of text in an *8½* window. Under each bitmap is displayed one or two lines of text presenting parameters of the image. The first line shows the bitmap's `ldepth`, the log base 2 of the number of bits per pixel; `r`, the rectangle covered by the image; and the name of the `file` from which it was read. If the file is a subfont, a second line presents a hexadecimal Unicode `offset` to be applied to character values from the subfont (typically as stored in a font file; see *font*(6)); and the subfont's `n`, `height`, and `ascent` as defined in *cachechars*(2).

By means described below, magnified views of portions of the bitmaps may be displayed. The text associated with such a view includes `mag`, the magnification. If the view is of a single character from a subfont, the second line of text shows the character's value (including the subfont's offset) in hexadecimal and as a character in *tweak's* default font; the character's `x`, `top`, `bottom`, `left`, and `width` as defined in *cachechars*(2); and `iwidth`, the physical width of the image in the subfont's bitmap.

There are two methods to obtain a magnified view of a character from a subfont. The first is to click mouse button 1 over the image of the character in the subfont. The second is to select the `char` entry on the button 3 menu, point the resulting gunsight cursor at the desired subfont and click button 3, and then type at the text prompt at the bottom of the screen the character value, either as a multi-digit hexadecimal number or as a single rune representing the character.

To magnify a portion of other types of bitmap files, click button 1 over the unmagnified file. The cursor will switch to a cross. Still with button 1, sweep a rectangle, as in `8½`, that encloses the portion of the image to be magnified.

Depressing buttons 1 and 2 change within magnified images changes pixel values. By default, button 1 sets the pixel to all ones and button 2 sets the pixel to all zeros.

Across the top of the screen is a textual display of global parameters. These values, as well as many of the textual values associated with the images, may be edited by clicking button 1 on the displayed value and typing a new value. The values along the top of the screen are:

`mag`	Default magnification.
`val(hex)`	The value used to modify pixels within magnified images. The value must be in hexadecimal, optionally preceded by a tilde for bitwise negation.
`but1`	
`but2`	The boolean function used by the named button to set pixel values. The function may be specified either by name as defined in `<libg.h>`, e.g. `DorS`, or by simple boolean expression, e.g. `S|D`. In these expressions, `S` is the pixel value defined above and `D` is the pixel being modified.
`copy`	The boolean function used in the `copy` menu item.

Under button 3 is a menu holding a variety of functions. Many of these functions prompt for the image upon which to act by switching to a gunsight cursor; click button 3 over the selection, or click a different button to cancel the action.

`open`	Read and display a file. The name of the file is typed to the prompt on the bottom line.
`read`	Reread a file.
`write`	Write a file.

`copy`	Use the copy function, default `S`, to transfer a rectangle of pixels from one image to another. The program prompts with a cross cursor; sweep out a rectangle in one image or just click button 3 to select the whole image. The program will leave that rectangle in place and attach another one to the cursor. Move that rectangle to the desired place in any image and click button 3, or another button to cancel the action.
`char`	As described above, open a magnified view of a character image in a subfont.
`close`	Close the specified image. If the image is the unmagnified file, also close any magnified views of that file.
`exit`	Quit *tweak.* The program will complain once about modified but unwritten files.

To clear blocks of pixels, use `copy` with function `0`.

SEE ALSO

art(1), *bitmap*(6)

NAME

twig − tree-manipulation language

SYNOPSIS

`twig` [`-sASC`] [`-w` *suffix*] *file.mt*

DESCRIPTION

Twig converts a tree-specification scheme consisting of pattern-action rules with associated costs into C functions that can be called to manipulate input trees. The C functions first find a minimum-cost covering of an input tree using a dynamic programming algorithm and then execute the actions associated with the patterns used in the covering. The tree-specification scheme may allow several coverings for an input tree, but the dynamic programming algorithm resolves any ambiguities by selecting a cheapest covering.

The input file containing the tree-specification scheme must have the suffix `.mt`. *Twig* produces two output files: `walker.c`, which becomes the source file for the tree matcher, and `symbols.h`, which contains the definitions for the node and label symbols used in the source file.

To build `walker.c`, *twig* uses an internal template file, by default on appropriate for use with *fprintf*(2). The options are

`-A`	Use a template file for ANSI/POSIX source files.
`-C`	Use a template file for files that use libc's *print*(2) routines.
`-S`	Use a template file for files that use *fprintf*(2).
`-s`	Do not produce a `symbols.h` file.
`-w` *suffix*	Use the template file `/sys/lib/twig/walker.`*suffix.*

FILES

`/sys/lib/twig`	System area for template files.

SEE ALSO

yacc(1)

S. W. K. Tjiang, *The Twig Reference Manual,* Computing Science Technical Report No. 120, AT&T Bell Laboratories, Murray Hill, N.J.

A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, *Code generation using tree matching and dynamic programming.*

BUGS

When tree matching fails, the debugging output is cryptic.

NAME

uniq − report repeated lines in a file

SYNOPSIS

`uniq` [`-udc` [`+-`*num*]] [*file*]

DESCRIPTION

Uniq copies the input *file,* or the standard input, to the standard output comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed. Repeated lines must be adjacent in order to be found.

`-u`	Print unique lines.
`-d`	Print (one copy of) duplicated lines.
`-c`	Prefix a repetition count and a tab to each output line. Implies `-u` and `-d`.
−*num*	The first *num* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.
+*num*	The first *num* characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(1)

BUGS

Field-selection and comparison should be compatible with *sort*(1).

NAME

units − conversion program

SYNOPSIS

`units` [*file*]

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
you have: inch
you want: cm
    * 2.54
    / 0.393701
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
you have: 15 pounds force/in2
you want: atm
    * 1.02069
    / .97973
```

Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

`pi`	ratio of circumference to diameter
`c`	speed of light
`e`	charge on an electron
`g`	acceleration of gravity
`force`	same as `g`
`mole`	Avogadro's number
`water`	pressure head per unit height of water
`au`	astronomical unit

The `pound` is a unit of mass. Compound names are run together, e.g. `lightyear`. British units that differ from their US counterparts are prefixed thus: `brgallon`. Currency is denoted `belgiumfranc`, `britainpound`, etc.

The complete list of units can be found in `/lib/units`. A *file* argument to *units* specifies a file to be used instead of `/lib/units`.

FILES

`/lib/units`

BUGS

Since *units* does only multiplicative scale changes, it can convert Kelvin to Rankine but not Centigrade to Fahrenheit.
Currency conversions are only as accurate as the last time someone updated `/lib/units`.

NAME

vi, ki − instruction simulators

SYNOPSIS

`vi` [*textfile*]
`vi` *pid*
`ki` [*textfile*]
`ki` *pid*

DESCRIPTION

Vi simulates the execution of a MIPS binary in a Plan 9 environment. It has two main uses: as a debugger and as a statistics gatherer. Programs running under *vi* execute about two hundred times slower than normal—but faster than single stepping under *db. Ki* is similar to *vi* but interprets SPARC binaries. The following discussion refers to *vi* but applies to *ki* as well.

Vi will simulate the execution of a named *textfile.* It will also make a copy of an existing process with process id *pid* and simulate its continuation.

As a debugger *vi* offers more complete information than *db*(1). Tracing can be performed at the level of instructions, system calls, or function calls. *Vi* allows breakpoints to be triggered when specified addresses in memory are accessed. A report of instruction counts, load delay fills and distribution is produced for each run. *Vi* simulates the CPU's caches and MMU to assist the optimization of compilers and programs.

The command interface mirrors the interface to *db*; see *db*(1) for a detailed description. Data formats and addressing are compatible with *db* except for disassembly: *vi* offers only MIPS (`db -mmipsco`) mnemonics for machine instructions. *Ki* offers both Plan 9 and Sun SPARC formats.

Several extra commands allow extended tracing and printing of statistics:

`$t[0ics]`
The *t* command controls tracing. Zero cancels all tracing options.

`i` Enable instruction tracing

`c` Enable call tracing

`s` Enable system call tracing

`$i[itsp]`
The `i` command prints statistics accumulated by all code run in this session.

`i` Print instruction counts and frequency.

`p` Print cycle profile.

`t` (*Vi* only) Print TLB and cache statistics.

`s` Print memory reference, working set and size statistics.

`:b[arwe]`
Vi allows breakpoints to be set on any memory location. These breakpoints monitor when a location is accessed, read, written, or equals a certain value. For equality the compared value is the *count* (see *db*(1)) supplied to the command.

SEE ALSO

nm(1), *db*(1)

BUGS

The code generated by `vc` and `kc` are well supported, but some unusual instructions are unimplemented. Some Plan 9 system calls such as *fork* cause simulated traps. The floating point simulation makes assumptions about the underlying machine floating point support. The floating point conversions performed by vi may cause a loss of precision.

NAME

wc − word count

SYNOPSIS

`wc` [`-lwrbc`] [*file* ...]

DESCRIPTION

Wc counts lines, words, runes, syntactically-invalid UTF codes and bytes in the named *files,* or in the standard input if no file is named. A word is a maximal string of characters delimited by spaces, tabs or newlines. The count of runes includes invalid codes.

If the optional argument is present, just the specified counts (lines, words, runes, broken UTF codes or bytes) are selected by the letters `l`, `w`, `r`, `b`, or `c`. Otherwise, lines, words and bytes (`-lwc`) are reported.

BUGS

Unicode has many blank characters scattered through it, but *wc* looks for only ASCII space, tab and newline.

Wc should have options to count suboptimal utf codes and bytes that cannot occur in any utf code.

NAME

who, whois − who is using the machine

SYNOPSIS

`who`

`whois` *person*

DESCRIPTION

Who prints the name of everyone with a non-Exiting process on the current machine.

Whois looks in `/adm/whois` and `/adm/users` to find out more information about *person.*

NAME

xd − hex, octal, decimal, or ASCII dump

SYNOPSIS

`xd` [*option ...*] [*−format ...*] [*file ...*]

DESCRIPTION

Xd concatenates and dumps the *files* (standard input by default) in one or more formats. Groups of 16 bytes are printed in each of the named formats, one format per line. Each line of output is prefixed by its address (byte offset) in the input file. The first line of output for each group is zero-padded; subsequent are blank-padded.

Formats other than `−c` are specified by pairs of characters telling size and style, `4x` by default. The sizes are

`1` or `b`	1-byte units.
`2` or `w`	2-byte big-endian units.
`4` or `l`	4-byte big-endian units.

The styles are

`o`	Octal.
`x`	Hexadecimal.
`d`	Decimal.

Other options are

`−c`	Format as `1x` but print ASCII representations or C escape sequences where possible.
`−a`*style*	Print file addresses in the given style (and size 4).
`−u`	(Unbuffered) Flush the output buffer after each 16-byte sequence.
`−s`	Reverse (swab) the order of bytes in each group of 4 before printing.
`−r`	Print repeating groups of identical 16-byte sequences as the first group followed by an asterisk.

SEE ALSO

db(1)

BUGS

The various output formats don’t line up properly in the output of *xd.*

NAME

yacc − yet another compiler-compiler

SYNOPSIS

`yacc` [*option ...*] *grammar*

DESCRIPTION

Yacc converts a context-free grammar and translation code into a set of tables for an LR(1) parser and translator. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, `y.tab.c`, must be compiled by the C compiler to produce a program `yyparse`. This program must be loaded with a lexical analyzer function, `yylex(void)` (often generated by *lex*(1)), with a `main(int argc, char *argv[])` program, and with an error handling routine, `yyerror(char*)`.

The options are

`-o` *output*	Direct output to the specified file instead of `y.tab.c.`
`-D`*n*	Create file `y.debug`, containing diagnostic messages. The amount of diagnostic output from the parser is regulated by value *n*: 0 Report errors. 1 Also report reductions. 2 Also report the name of each token returned by `yylex`.
`-v`	Create file `y.output`, containing a description of the parsing tables and of conflicts arising from ambiguities in the grammar.
`-d`	Create file `y.tab.h`, containing `#define` statements that associate *yacc*-assigned ‘token codes’ with user-declared ‘token names’. Include it in source files other than `y.tab.c` to give access to the token codes.
`-s` *stem*	Change the prefix `y` of the file names `y.tab.c`, `y.tab.h`, `y.debug`, and `y.output` to *stem.*
`-S`	Write a parser that uses *stdio* instead of the *print* routines in libc.

FILES

`y.output`	
`y.tab.c`	
`y.tab.h`	
`y.debug`	
`y.tmp.*`	temporary file
`y.acts.*`	temporary file
`/sys/lib/yaccpar`	parser prototype
`/sys/lib/yaccpars`	parser prototype using stdio

SEE ALSO

lex(1)
Yacc: A parser generator, S. C. Johnson and R. Sethi, Unix Research System Programmer’s Manual, Volume 2

BUGS

The parser may not have full information when it writes to `y.debug` so that the names of the tokens returned by `yylex` may be missing.

NAME

yesterday − print file names from the dump

SYNOPSIS

`yesterday` [`-c`] [−*date*] *files ...*

DESCRIPTION

Yesterday prints the names of the *files* from the most recent dump. Since dumps are done early in the morning, yesterday's files are really in today's dump. For example, if today is March 17, 1992,

```
yesterday /adm/users
```

prints

```
/n/dump/1992/0317/adm/users
```

In fact, the implementation is to select the most recent dump in the current year, so the dump selected may not be from today.

With option `-c`, *yesterday* copies the dump file to the current directory.

The *date* option selects other day's dumps, with a format of 2, 4, 6, or 8 digits of the form *dd, mmdd, yymmdd,* or *yyyymmdd.*

Yesterday does not guarantee that the string it prints represents an existing file.

EXAMPLES

Back up to yesterday's MIPS binary of `vc`:

```
cd /mips/bin
yesterday -c vc
```

Temporarily back up to March 1's MIPS C library to see if a program runs correctly when loaded with it:

```
bind `{yesterday -0301 /mips/lib/libc.a} /mips/lib/libc.a
rm v.out
mk
v.out
```

FILES

`/n/dump`

SEE ALSO

fs(4)

BUGS

It's hard to use this command without singing.

NAME

intro − introduction to library functions

SYNOPSIS

```
#include <u.h>

#include <libc.h>

#include <stdio.h>

#include <bio.h>

#include <libg.h>

#include <gnot.h>

#include <frame.h>

#include <layer.h>

#include <regexp.h>
```

DESCRIPTION

This section describes functions in various libraries. For the most part, each library is defined by a single C include file, listed above, and a single archive file containing the library proper. The name of the archive is `/$objtype/lib/lib`*x*`.a`, where *x* is the base of the include file name, stripped of a leading `lib` if present. For example, `<libg.h>` defines the contents of library `/$objtype/lib/libg.a` which may be abbreviated when named to the loader as `-lg` . In practice, each include file contains a `#pragma` that directs the loader to pick up the associated archive automatically, so it is rarely necessary to tell the loader which libraries a program needs.

The library to which a function belongs is identified by the section number at the top of the manual page:

(2) These functions constitute the 'C library', *libc,* containing most of the basic non-system call subroutines such as *strlen* . Declarations for all of these functions are in `<libc.h>`, which must be preceded by (*needs*) an include of `<u.h>`.

(2G) These functions constitute the library *libg*, the graphics library. Declarations for these functions are in `<libg.h>`, which needs `<libc.h>` and `<u.h>`.

(2S) These functions constitute the library *libstdio*, the 'standard I/O package' (see *fgetc*(2)). Declarations for these functions are in `<stdio.h>`.

(2X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

The include file `<u.h>`, a prerequisite of several other include files, declares the architecture-dependent and -independent types, including: *ushort*, *uchar*, and *ulong*, the unsigned integer types; *schar*, the signed char type; *vlong*, a very long integral type; *jmp_buf*, the type of the argument to *setjmp* and *longjmp,* plus macros that define the layout of *jmp_buf* (see *setjmp*(2)); definitions of the bits in the floating-point control register as used by *getfcr*(2); *Length*, a union giving different views of the 64-bit length of a file, declared as

```
typedef union
{
     char   clength[8];
     vlong  vlength;
     struct
     {
            long hlength;   /* high order */
            long length;    /* low order */
     };
} Length;
```

Name space

Files are collected into a hierarchical organization called a *file tree* starting in a *directory* called the *root.* File names, also called *paths*, consist of a number of `/`-separated *path elements* with the slashes corresponding to directories. A path element must contain only printable characters that occupy no more than `NAMELEN-1` bytes. A path element cannot contain a space or slash.

When a process presents a file name to Plan 9, it is *evaluated* by the following algorithm. Start with a directory that depends on the first character of the path: `/` means the root of the main hierarchy, `#` means the separate root of a kernel device's file tree (see Section 3), and anything else means the process's current working directory. Then for each path element, look up the element in the directory, advance to that directory, do a possible translation (see below) and repeat. The last step may yield a directory or regular file. The collection of files reachable from the root is called the *name space* of a process.

A program can use *bind* or *mount* (see *bind*(2)) to say that whenever a specified file is reached during evaluation, evaluation instead continues from a second specified file. Also, the same system calls create *union directories*, which are concatenations of ordinary directories that are searched sequentially until the desired element is found. Using *bind* and *mount* to do name space adjustment affects only the current process group (see below). Certain conventions about the layout of the name space should be preserved; see *namespace*(4).

File I/O

Files are opened for input or output by *open* or *create* (see *open*(2)). These calls return an integer called a *file descriptor* which identifies the file to subsequent I/O calls, notably *read*(2) and *write*. File descriptors range from 0 to 99 in the current system. The system allocates the numbers by selecting the lowest unused descriptor. They may be reassigned using *dup*(2). File descriptors are indices into a kernel resident *file descriptor table*. Each process has an associated file descriptor table. In some cases (see *rfork* in *fork*(2)) a file descriptor table may be shared by several processes.

By convention, file descriptor 0 is the standard input, 1 is the standard output, and 2 is the standard error output. With one exception, the operating system is unaware of these conventions; it is permissible to close file 0, or even to replace it by a file open only for writing, but many programs will be confused by such chicanery. The exception is that the system prints messages about broken processes to file descriptor 2.

Files are normally read or written in sequential order. The I/O position in the file is called the *file offset* and may be set arbitrarily using the *seek*(2) system call.

Directories may be opened and read much like regular files. They contain an integral number of records, called *directory entries,* of length `DIRLEN` (defined in `<libc.h>`). Each entry is a machine-independent representation of the information about an existing file in the directory, including the name, ownership, permission, access dates, and so on. The entry corresponding to an arbitrary file can be retrieved by *stat*(2) or *fstat*; *wstat* and *fwstat* write back entries, thus changing the properties of a file. An entry may be translated into a more convenient, addressable form called a `Dir` structure; *dirstat, dirfstat, dirwstat,* and *dirfwstat* execute the appropriate translations (see *stat*(2)).

New files are made with *create* (in *open*(2)) and deleted with *remove*(2). Directories may not directly be written; *create*, *remove*, *wstat*, and *fwstat* alter them.

Pipe(2) creates a connected pair of file descriptors, useful for local communication.

Process execution and control

A new process is created when an existing one calls *rfork* with the `RFPROC` bit set, usually just by calling *fork*(2). The new (child) process starts out with copies of the address space and most other attributes of the old (parent) process. In particular, the child starts out running the same program as the parent; *exec*(2) will bring in a different one.

Each process has a unique integer process id; a set of open files, indexed by file descriptor; and a current working directory (changed by *chdir*(2)).

Each process has a set of attributes — memory, open files, name space, etc. — that may be shared or unique. Flags to *rfork* control the sharing of these attributes.

A process terminates by calling *exits*(2). A parent process may call *wait* (in *exits*(2)) to wait for some child to terminate. A string of status information may be passed from *exits* to *wait*. A process can go to sleep for a specified time by calling *sleep*(2).

There is a *notification* mechanism for telling a process about events such as address faults, floating point faults, and messages from other processes. A process uses *notify*(2) to register the function to be called (the *notification handler*) when such events occur.

SEE ALSO

nm(1), *2l*(1), *2c*(1)

DIAGNOSTICS

Math functions in *libc* will return special values when the function is undefined for the given arguments or when the value is not representable (see *nan*(2)).

Some of the functions in *libc* are system calls and many others employ system calls in their implementation. All system calls return integers, with −1 indicating that an error occurred; *errstr*(2) recovers a string describing the error. Functions that may affect the value of the error string are said to ‘‘set *errstr*’’; it is understood that the error string is altered only if an error occurs.

NAME

abort − generate a fault

SYNOPSIS

```
void  abort(void)
```

DESCRIPTION

Abort causes an access fault, causing the current process to enter the ‘Broken’ state. The process can then be inspected by a debugger.

NAME

abs, labs − integer absolute values

SYNOPSIS

```
int   abs(int a)

long  labs(long a)
```

DESCRIPTION

Abs returns the absolute value of integer *a*, and *labs* does the same for a long.

SEE ALSO

floor(2) for *fabs*

DIAGNOSTICS

Abs and *labs* return the most negative integer or long when the true result is unrepresentable.

NAME

access − determine accessibility of file

SYNOPSIS

```
int access(char *name, int mode)
```

DESCRIPTION

Access evaluates the given file *name* for accessibility. If *mode*`&4` is nonzero, read permission is expected; if *mode*`&2`, write permission; if *mode*`&1`, execute permission. If *mode*`==0`, the file merely need exist. In any case all directories leading to the file must permit searches. Zero is returned if the desired access is permitted, −1 if not.

Only access bits are checked. A directory may be announced as writable by *access*, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but *exec*(2) will fail unless it is in proper format.

SEE ALSO

stat(2)

DIAGNOSTICS

Sets *errstr.*

NAME

add, sub, mul, div, raddp, rsubp, rmul, rdiv, rshift, inset, rcanon, eqpt, eqrect, ptinrect, rectinrect, rectXrect, rectclip, Dx, Dy, Pt, Rect, Rpt − arithmetic on points and rectangles

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>

Point     add(Point p, Point q)
Point     sub(Point p, Point q)
Point     mul(Point p, int a)
Point     div(Point p, int a)
Rectangle raddp(Rectangle r, Point p)
Rectangle rsubp(Rectangle r, Point p)
Rectangle rmul(Rectangle r, int a)
Rectangle rdiv(Rectangle r, int a)
Rectangle rshift(Rectangle r, int a)
Rectangle inset(Rectangle r, int n)
Rectangle rcanon(Rectangle r)
int       eqpt(Point p, Point q)
int       eqrect(Rectangle r, Rectangle s)
int       ptinrect(Point p, Rectangle r)
int       rectinrect(Rectangle r, Rectangle s)
int       rectXrect(Rectangle r, Rectangle s)
int       rectclip(Rectangle *rp, Rectangle b)
int       Dx(Rectangle r)
int       Dy(Rectangle r)
Point     Pt(int x, int y)
Rectangle Rect(int x0, int y0, int x1, int y1)
Rectangle Rpt(Point p, Point q)
```

DESCRIPTION

The functions *Pt*, *Rect* and *Rpt* construct geometrical data types from their components. These are implemented as macros.

Add returns the Point sum of its arguments: `Pt(p.x+q.x, p.y+q.y)`. *Sub* returns the Point difference of its arguments: `Pt(p.x−q.x, p.y−q.y)`. *Mul* returns the Point `Pt(p.x*a, p.y*a)`. *Div* returns the Point `Pt(p.x/a, p.y/a)`.

Raddp returns the Rectangle `Rect(add(r.min, p), add(r.max, p))`; *rsubp* returns the Rectangle `Rpt(sub(r.min, p), sub(r.max, p))`. *Rmul* returns the Rectangle `Rpt(mul(r.min,a), mul(r.max,a))`; *Rdiv* returns the Rectangle `Rpt(div(r.min,a), div(r.max,a))`.

Rshift returns the rectangle *r* with all coordinates either left-shifted or right-shifted by *a*, depending on whether *a* is positive or negative, respectively.

Inset returns the Rectangle `Rect(r.min.x+n, r.min.y+n, r.max.x−n, r.max.y−n)`.

Rcanon returns a rectangle with the same extent as *r*, canonicalized so that `min.x` ≤ `max.x`, and `min.y` ≤ `max.y`.

Eqpt compares its argument Points and returns 0 if unequal, 1 if equal. *Eqrect* does the same for its argument Rectangles.

Ptinrect returns 1 if *p* is a point within *r*, and 0 otherwise.

Rectinrect returns 1 if all the pixels in *r* are also in *s*, and 0 otherwise.

RectXrect returns 1 if *r* and *s* share any point, and 0 otherwise.

Rectclip clips in place the Rectangle pointed to by *rp* so that it is completely contained within *b*. The return value is 1 if any part of **rp* is within *b*. Otherwise, the return value is 0 and **rp* is unchanged.

The functions *Dx* and *Dy* give the width (x) and height (y) of a Rectangle. They are implemented as macros.

SEE ALSO

graphics(2)

NAME

ARG – process option letters from argv

SYNOPSIS

```
#include <libc.h>
ARGBEGIN {
} ARGEND
char *ARGF();
Rune ARGC();

extern char *argv0;
```

DESCRIPTION

Command-line arguments to programs (see *exec*(2)) are conventionally arranged as a set of *options* — strings beginning with a `-` (minus sign) — followed by plain arguments such as file names. The `ARG` macros provide a convenient means for processing these options. Option characters appear in nonempty clusters preceded by `-`. After options processing is terminated (by an argument not starting with `-`, or by the argument `--` which is then skipped, or by the argument `-`), execution resumes after the `ARGEND`.

The body of a `switch` statement should be put between `ARGBEGIN{` and `}ARGEND`; it is executed once for each option character (the character itself may be referenced by `ARGC()`). If an option takes a string argument (that is, the rest of the current option string or if that is empty, the next argument), it can be referenced by `ARGF()`. After `ARGEND`, *argv* points at a zero-terminated list of the remaining *argc* arguments.

`ARGBEGIN` also sets up *argv0* to point at `argv[0]` (conventionally the name of the program).

EXAMPLES

This program processes arguments for a command that can take option `b` and option `f`, which requires an argument.

```
#include <u.h>
#include <libc.h>

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

        print("%s", argv[0]);
        ARGBEGIN{
        case 'b':       print(" -b"); break;
        case 'f':       f = ARGF(); print(" -f(%s)", f?f:"no arg"); break;
        default:        print(" badflag('%c')", ARGC()); break;
        }ARGEND
        print(" %d args:", argc);
        while(*argv)
                print(" '%s'", *argv++);
        print("\n");
        exits(0);
}
```

When this program is run as

```
        prog -bffile1 -r -f file2 arg1 argn
```

it yields

```
        prog -b -f(file1) badflag('r') -f(file2) 2 args: 'arg1' 'argn'
```

DIAGNOSTICS

`ARGF()` returns 0 on error.

NAME

atof, atoi, atol, charstod, strtod, strtol, strtoul − convert text to numbers

SYNOPSIS

```
double atof(char *nptr)

int    atoi(char *nptr)

long   atol(char *nptr)

double charstod(int (*f)(void *), void *a)

double strtod(char *nptr, char **rptr)

long   strtol(char *nptr, char **rptr, int base)

ulong  strtoul(char *nptr, char **rptr, int base)
```

DESCRIPTION

Atof, *atoi*, and *atol* convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string. Leading C escapes are understood, as in *strtol* with *base* zero.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e` or `E` followed by an optionally signed integer.

Atoi and *atol* recognize an optional string of tabs and spaces, then an optional sign, then a string of decimal digits.

Strtod, strtol, and *strtoul,* behave similarly to *atof,* and *atol* and, if *rptr* is not zero, set **rptr* to point to the input character immediately after the string converted.

Strtol and *strtoul* interpret the digit string in the specified *base,* from 2 to 36, each digit being less than the base. Digits with value over 9 are represented by letters, a-z or A-Z. If *base* is 0, the input is interpreted as an integral constant in the style of C (with no suffixed type indicators): numbers are octal if they begin with `0`, hexadecimal if they begin with `0x` or `0X`, otherwise decimal. *Strtoul* does not recognize signs.

Charstod interprets floating point numbers like *atof*, but it gets successive characters by calling `(*`*f*`)(a)`. The last call to *f* terminates the scan, so it must have returned a character that is not a legal continuation of a number. Therefore, it may be necessary to back up the input stream one character after calling *charstod*.

SEE ALSO

fscanf(2)

DIAGNOSTICS

Zero is returned if the beginning of the input string is not interpretable as a number; even in this case, *rptr* will be updated.

BUGS

Atoi and *atol* accept octal and hexadecimal numbers in the style of C, contrary to the ANSI specification.

NAME

auth, srvauth, getchall, challreply, newns, authdial, passtokey, nvcsum − network authentication

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <auth.h>

char* auth(int fd, char *dialstring)

char* srvauth(char *user)

int   getchall(char *user, char chall[NETCHLEN]);

int   challreply(int fd, char *user, char *response);

char* newns(char *user, char *nsfile)

int   authdial(char *service)

int   passtokey(char key[DESKEYLEN], char *password)

uchar nvcsum(void *mem, int len)
```

DESCRIPTION

Auth and *srvauth* authenticate connections for Plan 9 remote execution using the *rexauth* protocol described in *auth*(6). *Auth* authenticates an outgoing network call. *Fd* is a file descriptor to the data channel of the network connection. *Auth* extracts from *dialstring* the name of the server being called. *Dialstring* should be the address passed to *dial*(2). *Auth* reads the user's name with *getuser*(2) and uses `#c/crypt` for encrypting and decrypting *rexauth* messages.

Srvauth authenticates the corresponding incoming call. It copies the name of the user into *user*, which must be at least `NAMELEN` bytes long.

Getchall and *challreply* authenticate an incoming network call for a service that does not perform the usual Plan 9 authentication. They use the *chal* protocol described in *auth*(6). *User* points to the local name of the user. *Getchall* reads a null-terminated textual challenge from the authentication server and copies it to *chall*. It returns the open file descriptor to the authentication server, or −1 if it fails. The challenge should be printed for the user to see, and the user should use a Digital Pathways Securenet Key or *aux/netkey* (see *passwd*(1)) to generate the appropriate response.

Challreply should be called with the user's response, which is also a null-terminated text string, and the file descriptor returned from *getchall*. It returns 0 if it succeeds, or −1 if the user was not authenticated.

Srvauth and *challreply* set the process's user name and encryption key (see *cons*(3)).

Newns builds a name space for *user*. It opens the file *nsfile* (`/lib/namespace` is used if *nsfile* is null), copies the old environment, and erases the current name space, sets the environment variables `user` and `home`, and interprets the commands in *nsfile*. The format of *nsfile* is described in *namespace*(6).

Authdial calls *service* on the local authentication server. It returns a file descriptor to the open connection or −1 if it fails.

Passtokey converts *password* into a DES key and stores the result in *key*. It returns 0 if *password* could not be converted, and 1 otherwise.

Nvcsum computes a checksum for the *len* byte array *mem*. It is used to checksum keys stored in non-volatile RAM.

FILES

`#c/crypt`	Encryption file used by *auth*.
`/lib/namespace`	Default name space specification file.

DIAGNOSTICS

Auth, *srvauth*, and *newns* return a pointer to an error message upon failure, and 0 upon success.

SEE ALSO

passwd(1), *auth*(6), *cons*(3), *dial*(2)

NAME

balloc, bfree, rdbitmap, wrbitmap, rdbitmapfile, wrbitmapfile − allocating, freeing, reading, writing bitmaps

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>

Bitmap *balloc(Rectangle r, int ldepth)

void   bfree(Bitmap *b)

void   rdbitmap(Bitmap *b, int ymin, int ymax, uchar *data)

void   wrbitmap(Bitmap *b, int ymin, int ymax, uchar *data)

Bitmap *rdbitmapfile(int fd)

void   wrbitmapfile(int fd, Bitmap *b)
```

DESCRIPTION

A new bitmap is allocated with `balloc`; it will have the extent and *ldepth* (log base 2 of the number of bits per pixel) given by its arguments, and will be filled with zeros. The *id* field will have been set to the identifying number used by `/dev/bitblt` (see *bit*(3)), and the *cache* field will be zero. *Balloc* returns 0 if the server has run out of bitmap resources. *Bfree* frees the resources used by its argument bitmap.

The remaining functions deal with moving groups of pixel values between bitmaps and user space or external files. There is a fixed format for the exchange and storage of bitmap data (see *bitmap*(6)).

Rdbitmap reads rows of pixels from bitmap *b* into *data*. The rows read have *y*=*ymin*,*ymin*+1, ... *ymax*−1. Those rows must be within the range allowed by *b*`.r`.

`Wrbitmap` replaces the specified rows of pixels in bitmap *b* with *data*.

Rdbitmapfile creates a bitmap from data contained an external file (see *bitmap*(6) for the file format); *fd* is a file descriptor obtained by opening such a file for reading. The returned bitmap is allocated using *balloc.*

Wrbitmapfile writes bitmap *b* onto file descriptor *fd*, which should be open for writing. The format is as described for *rdbitmapfile*.

Rdbitmapfile and *wrbitmapfile* do not close *fd*.

SEE ALSO

graphics(2), *bitblt*(2), *bit*(3), *bitmap*(6)

DIAGNOSTICS

These functions return 0 on failure, usually due to insufficient memory.

May set *errstr.*

BUGS

`Ldepth` must be 0, 1, 2, or 3.

NAME

bind, mount, unmount − change name space

SYNOPSIS

```
int bind(char *name, char *old, int flag)

int mount(int fd, char *old, int flag, char *aname, char *authserv)

int unmount(char *name, char *old)
```

DESCRIPTION

Bind and *mount* modify the file name space of the current process and other processes in its name space group (see *fork*(2)). For both calls, *old* is the name of an existing file or directory in the current name space where the modification is to be made. The name *old* is *evaluated* as described in *intro*(2), except that no translation of the final path element is done.

For *bind*, *name* is the name of another (or possibly the same) existing file or directory in the current name space. After a successful *bind* call, the file name *old* is an alias for the object originally named by *name*; if the modification doesn't hide it, *name* will also still refer to its original file. The evaluation of *new* happens at the time of the *bind*, not when the binding is later used.

The *fd* argument to *mount* is a file descriptor of an open network connection or pipe to a file server. The *old* file must be a directory. After a successful *mount* the file tree *served* (see below) by *fd* will be visible with its root directory having name *old*.

The *flag* controls details of the modification made to the name space. In the following, *new* refers to the file as defined by *name* or the root directory served by *fd*. Either both *old* and new files must be directories, or both must not be directories. *Flag* can be one of:

`MREPL`	Replace the *old* file by the new one. Henceforth, an evaluation of *old* will be translated to the new file. If they are directories (for *mount*, this condition is true by definition), *old* becomes a *union directory* consisting of one directory (the new file).
`MBEFORE`	Both the *old* and new files must be directories. Add the constituent files of the new directory to the union directory at *old* so its contents appear first in the union. After an `MBEFORE` *bind* or *mount,* the new directory will be searched first when evaluating file names in the union directory.
`MAFTER`	Like `MBEFORE` but the new directory goes at the end of the union.

The flags are defined in `<libc.h>`. In addition, there is an `MCREATE` flag that can be OR'd with any of the above. When a *create* system call (see *open*(2)) attempts to create in a union directory, and the file does not exist, the elements of the union are searched in order until one is found with `MCREATE` set. The file is created in that directory; if that attempt fails, the *create* fails.

With *mount,* the file descriptor *fd* must be open for reading and writing and prepared to respond to 9P messages (see Section 5). After the *mount,* the file tree starting at *old* is served by a kernel *mnt*(3) device. That device will turn operations in the tree into messages on *fd*. *Aname* selects among different file trees on the server; the null string chooses the default tree. *Authserv* is the textual name of the file server. It is used during authentication to assure that *fd* is a connection to the intended server. If *authserv* is the empty string, no authentication is performed and the empty string is used as the authentication key in the `attach` message (see *auth*(5) and *attach*(5)).

The file descriptor *fd* is automatically closed by a successful *mount* call.

The effects of *bind* and *mount* can be undone by *unmount.* If *name* is zero, everything bound to or mounted upon *old* is unbound or unmounted. If *name* is not zero, it is evaluated as described above for *bind*, and the effect of binding or mounting that particular result on *old* is undone.

SEE ALSO

bind(1), *intro*(2), *fcall*(2), *intro*(5), *mnt*(3), *srv*(3)

DIAGNOSTICS

These routines set *errstr*.

BUGS

Mount will not return until it has successfully attached to the file server, so the process doing a *mount* cannot be the one serving.

NAME

Bopen, Binit, Binits, Brdline, Bgetc, Bgetrune, Bgetd, Bungetc, Bungetrune, Bread, Bseek, Boffset, Bfildes, Blinelen, Bputc, Bputrune, Bprint, Bwrite, Bflush, Bclose, Bbuffered − buffered input/output

SYNOPSIS

```
#include <bio.h>

Biobuf*Bopen(char *file, int mode)

int   Binit(Biobuf *bp, int fd, int mode)

int   Binits(Biobufhdr *bp, int fd, int mode, uchar *buf, int size)

int   Bclose(Biobufhdr *bp)

int   Bprint(Biobufhdr *bp, char *format, ...)

void* Brdline(Biobufhdr *bp, int delim)

int   Blinelen(Biobufhdr *bp)

long  Boffset(Biobufhdr *bp)

int   Bfildes(Biobufhdr *bp)

int   Bgetc(Biobufhdr *bp)

long  Bgetrune(Biobufhdr *bp)

int   Bgetd(Biobufhdr *bp, double *d)

int   Bungetc(Biobufhdr *bp)

int   Bungetrune(Biobufhdr *bp)

long  Bseek(Biobufhdr *bp, long offset, int ptr)

int   Bputc(Biobufhdr *bp, int c)

int   Bputrune(Biobufhdr *bp, long c)

long  Bread(Biobufhdr *bp, void *addr, long nbytes)

long  Bwrite(Biobufhdr *bp, void *addr, long nbytes)

int   Bflush(Biobufhdr *bp)

int   Bbuffered(Biobufhdr *bp)
```

DESCRIPTION

These routines implement fast buffered I/O. I/O on different file descriptors is independent.

Bopen opens *file* for mode `OREAD` or creates for mode `OWRITE`. It calls *malloc*(2) to allocate a buffer.

Binit initializes a standard size buffer, type *Biobuf,* with the open file descriptor passed in by the user. *Binits* initializes a non-standard size buffer, type *Biobufhdr,* with the open file descriptor, buffer area, and buffer size passed in by the user. *Biobuf* and *Biobufhdr* are related by the declaration:

```
typedef struct Biobuf Biobuf;
struct Biobuf
{
      Biobufhdr;
      uchar b[Bungetsize+Bsize];
};
```

Because of type promotion in our compiler, arguments of types pointer to Biobuf and pointer to Biobufhdr can be used interchangeably in the following routines.

Bopen, Binit, or *Binits* should be called before any of the other routines on that buffer. *Bfildes* returns the integer file descriptor of the associated open file.

Bclose flushes the buffer for *bp*. If the buffer was allocated by *Bopen*, the buffer is *freed* and the file is closed.

Brdline reads a string from the file associated with *bp* up to and including the first *delim* character. The delim character at the end of the line is not altered. *Brdline* returns a pointer to the start of the line or `0` on end-of-file or read error. *Blinelen* returns the length (including the delim) of the most recent string returned by *Brdline*.

Bgetc returns the next character from *bp*, or a negative value at end of file. *Bungetc* may be called immediately after *Bgetc* to allow the same character to be reread.

Bgetrune calls *Bgetc* to read the bytes of the next UTF sequence in the input stream and returns the value of the rune represented by the sequence. It returns a negative value at end of file. *Bungetrune* may be called immediately after *Bgetrune* to allow the same UTF sequence to be reread as either bytes or a rune. *Bungetc* and *Bungetrune* may back up a maximum of five bytes.

Bgetd uses *charstod* (see *atof*(2)) and *Bgetc* to read the formatted floating-point number in the input stream, skipping initial blanks and tabs. The value is stored in `*d.`

Bread reads *nbytes* of data from *bp* into memory starting at *addr*. The number of bytes read is returned on success and a negative value is returned if a read error occurred.

Bseek applies *seek*(2) to *bp*. It returns the new file offset. *Boffset* returns the file offset of the next character to be processed.

Bputc outputs the low order 8 bits of *c* on *bp*. If this causes a *write* to occur and there is an error, a negative value is returned. Otherwise, a zero is returned.

Bputrune calls *Bputc* to output the low order 16 bits of *c* as a rune in UTF format on the output stream.

Bprint is a buffered interface to *print*(2). If this causes a *write* to occur and there is an error, a negative value (`Beof`) is returned. Otherwise, the number of bytes output is returned.

Bwrite outputs *nbytes* of data starting at *addr* to *bp*. If this causes a *write* to occur and there is an error, a negative value is returned. Otherwise, the number of bytes written is returned.

Bflush causes any buffered output associated with *bp* to be written. The return is as for *Bputc*. *Bflush* is called on exit for every buffer still open for writing.

Bbuffered returns the number of bytes in the buffer. When reading, this is the number of bytes still available from the last read on the file; when writing, it is the number of bytes ready to be written.

The macros *BGETC*, *BPUTC*, *BOFFSET*, *BFILDES*, and *BLINELEN* are provided as fast versions of the corresponding routines.

SEE ALSO

open(2), *print*(2), *exits*(2), *utf*(6),

DIAGNOSTICS

Bio routines that return integers yield `Beof` if *bp* is not the descriptor of an open file. *Bopen* returns zero if the file cannot be opened in the given mode. All routines set *errstr* on error.

BUGS

Brdline returns an error on strings longer than the buffer associated with the file and also if the end-of-file is encountered before a delimiter. *Blinelen* will tell how many characters should be skipped in these cases. In the case of a true end-of-file, *Blinelen* will return zero.

The data returned by *Brdline* may be overwritten by calls to any other *bio* routine on the same *bp.*

NAME

bitblt, bitbltclip, clipline, point, segment, polysegment, arc, circle, disc, ellipse, texture, border, string, strsize, strwidth, Fcode − graphics functions

SYNOPSIS

```
#include <u.h>
#include <libg.h>

void  bitblt(Bitmap *db, Point dp, Bitmap *sb,
          Rectangle sr, Fcode f)

int   bitbltclip(void *)

int   clipline(Rectangle r, Point *p0, Point *p1)

void  point(Bitmap *b, Point p, int v, Fcode f)

void  segment(Bitmap *b, Point p, Point q, int v, Fcode f)

void  polysegment(Bitmap *b, int n, Point *pp, int v, Fcode f)

void  circle(Bitmap *b, Point p, int r, int v, Fcode f)

void  disc(Bitmap *b, Point p, int r, int v, Fcode)

void  arc(Bitmap *b, Point p0, Point p1, Point p2, int v, Fcode f)

void  ellipse(Bitmap *b, Point p, int a, int b, int v, Fcode f)

void  texture(Bitmap *b, Rectangle r, Bitmap *t, Fcode f)

void  border(Bitmap *b, Rectangle r, int w, Fcode f)

Point string(Bitmap *b, Point p, Font *ft, char *s, Fcode f)

Point strsize(Font *ft, char *s)

long  strwidth(Font *ft, char *s)

enum Fcode {
      Zero,      DnorS,    DandnotS, notS,
      notDandS,  notD,     DxorS,    DnandS,
      DandS,     DxnorS,   D,        DornotS,
      S,         notDorS,  DorS,     F,
} Fcode;
```

DESCRIPTION

Bitblt (bit-block transfer) takes bits from rectangle *sr* in the *source* Bitmap *sb* and overlays them on a congruent rectangle with the `min` corner at point *dp* in the *destination* bitmap, *db*. The *f* parameter defines each destination pixel as a function of the source and destination pixels. The sixteen codes in `Fcode` give all possible boolean operations on the source `S` and destination `D`. The code values may be expressed as boolean operations on the values `S` and `D`. For example, `D|S` computes the result as the logical *or* of the destination pixel's old value and the overlaying source pixel's value. If pixels are more than one bit deep, the operations are bitwise. The `Zero` and `F` codes result in new pixel values that are all zeros or all ones, respectively.

If the source and destination bitmaps have different depths, the source rectangle is first converted to have the same depth as the destination, as follows: conversion to a smaller number of bits per pixel is accomplished by taking the desired number of high order bits; conversion to a larger number of bits per pixel is accomplished by putting the small value into the high order bits, and replicating it as many times as necessary to fill the lower order bits.

All of the drawing graphics functions clip the rectangle against the source and destination bitmaps, so that only pixels within the destination bitmap are changed and none are changed that would have come from areas outside the source bitmap. *Bitbltclip* takes a pointer to the first argument of a *bitblt* argument list and clips *dp* and *sr* so the resulting *bitblt* is confined to the source and destination bitmaps. It returns one if the x and y dimensions of the resulting *bitblt* are positive; zero otherwise.

Point changes the value of the destination point *p* in bitmap *b* according to function code *f*. The source is a pixel with value *v*. The constant `~0` represents the maximum pixel value.

Segment, *circle*, *disc*, and *ellipse* all draw in bitmap *b* with function code *f* and a source pixel with value *v*. *Arc* draws a circular arc centered on *p0,* traveling clockwise from *p1* to *p2* or a point on the circle near *p2*. *Segment* draws a line segment in bitmap *b* from point *p* to *q*. The segment is half-open: *p* is the first point of the segment and *q* is the first point beyond the segment, so adjacent segments sharing endpoints abut. *Polysegment* draws the $n-1$ segments joining the *n* points in the array pointed to by *pp*. *Clipline* clips the line segment from **p0* to **p1* (*p0* is closed, *p1* is open) to rectangle *r*, adjusting *p0* and *p1* so that the segment is within the rectangle and **p1* is closed. It returns 0 if none of the segment is in the rectangle, 1 otherwise.

Circle draws a circle with radius *r* and center at point *p*. *Disc* is the same except that it fills the circle. *Ellipse* draws an ellipse with horizontal semi-axis *a* and vertical semi-axis *b*.

Border draws, with function *f* in bitmap *b*, the rectangular outline with lines of width *w* fitting just inside rectangle *r*.

Texture draws, with function *f* in bitmap *b*, a texture using the bitmap specified by *t*. The texture bitmap is aligned on *b*'s coordinate system so that (0,0) in both coordinate systems coincide, and then *t* is replicated to form a tiling of *b*. The tiling is clipped to rectangle *r* in *b*, and then transferred to *b* using the specified function.

String draws the text characters given by the null-terminated UTF string *s* into bitmap *b*, using font *ft.* The upper left corner of the first character (i.e., a point that is *ft*`->ascent` above the baseline) is placed at point *p*, and subsequent characters are placed on the same baseline, displaced to the right by the previous character's `width`. The individual characters are *bitblt*'ed into the destination, using drawing function *f*. *String* returns the point after the final character of *s*; this can be outside *b* if the string was clipped.

The bounding box for text to be drawn with *string* in font *ft* can be found with *strsize*; it returns the `max` point of the bounding box, assuming a `min` point of (0,0). *Strwidth* returns the *x*-component of the `max` point.

SEE ALSO

graphics(2), *utf*(6)

DIAGNOSTICS

These routines call the graphics error function on fatal errors.

NAME

brk, sbrk − change memory allocation

SYNOPSIS

```
int   brk(void *addr)

void* sbrk(ulong incr)
```

DESCRIPTION

Brk sets the system's idea of the lowest non-stack location not used by the program (called the break) to *addr* rounded up to the next multiple of 4 bytes. Locations not less than *addr* and below the stack pointer may cause a memory violation if accessed.

In the alternate function *sbrk, incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned. Rounding occurs as with *brk*.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *brk*. A call to *sbrk* with a zero argument returns the lowest address in the dynamic segment.

SEE ALSO

intro(2), *malloc*(2)

DIAGNOSTICS

These functions set *errstr.*

The error return from *sbrk* is `(void *)-1`.

NAME

cachechars, agefont, loadchar, Subfont, Fontchar, Font − font utilities

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>

int  cachechars(Font *f, char **s, ushort *c, int n, int *widp)

int  loadchar(Font *f, Rune r, Cacheinfo *c, int h, int noclr)

void agefont(Font *f)
```

DESCRIPTION

A *Font* may contain too many characters to hold in memory simultaneously. The graphics library and bitblt device (see *bit*(3)) cooperate to solve this problem by maintaining a cache of recently used character images. The details of this cooperation need not be known by most programs: *binit* and its associated *font* variable, *rdfontfile, charwidth, string,* and *ffree* are sufficient for most purposes. The routines described below are used internally by the graphics library to maintain the font cache.

A `Subfont` is a set of images for a contiguous range of characters, stored as a single bitmap with the characters placed side-by-side on a common baseline. It is described by the following data structures.

```
typedef
struct Fontchar {
     ushort   x;        /* left edge of bits */
     uchar    top;      /* first non-zero scan-line */
     uchar    bottom;   /* last non-zero scan-line */
     char     left;     /* offset of baseline */
     uchar    width;    /* width of baseline */
} Fontchar;

typedef
struct Subfont {
     short    n;        /* number of chars in subfont */
     char     height;   /* height of bitmap */
     char     ascent;   /* top of bitmap to baseline */
     Fontchar *info;    /* n+1 Fontchars */
     int      id;       /* id as known in /dev/bitblt */
} Font;
```

The bitmap fills the rectangle `(0, 0,` *w*`, height)`, where *w* is the sum of the horizontal extents (of non-zero pixels) for all characters. The pixels to be displayed for character *c* are in the rectangle `(`*i*`->x,` *i*`->top, (`*i*`+1)->x,` *i*`->bottom)` where *i* is `&subfont->info[`*c*`]`. When a character is displayed at `Point p` in a bitmap, the character rectangle is placed at `(p.x+`*i*`->left, p.y)` and the next character of the string is displayed at `(p.x+`*i*`->width, p.y)`. The baseline of the characters is `ascent` rows down from the top of the subfont bitmap. The `info` array has `n+1` elements, one each for characters 0 to `n−1` plus an additional entry so the size of the last character can be calculated. Thus the width, *w,* of the `Bitmap` associated with a `Subfont s` is `s->info[s->n].x`.

A `Font` consists of an overall height and ascent and a collection of subfonts together with the ranges of runes (see *utf*(6)) they represent. Fonts are described by the following structures.

```
typedef
struct Cachefont {
     Rune      min;      /* rune value of 0th char in subfont */
     Rune      max;      /* rune value+1 of last char in subfont */
     int       abs;      /* name has been made absolute */
```

```
     char      *name;
} Cachefont;

typedef
struct Cacheinfo {
     Rune      value;    /* value of character at this slot in cache */
     ushort    age;
     ulong     xright;   /* right edge of bits */
     Fontchar;
} Cacheinfo;

typedef
struct Cachesubf {
     ulong     age;      /* for replacement */
     Cachefont *cf;      /* font info that owns us */
     Subfont   *f;       /* attached subfont */
} Cachesubf;

typedef
struct Font {
     char      *name;
     uchar     height;   /* max height of bitmap, interline spacing */
     char      ascent;   /* top of bitmap to baseline */
     char      width;    /* widest so far; used in caching only */
     char      ldepth;   /* of images */
     short     id;       /* of font */
     short     nsub;     /* number of subfonts */
     ulong     age;      /* increasing counter; used for LRU */
     int       ncache;   /* size of cache */
     int       nsubf;    /* size of subfont list */
     Cacheinfo *cache;
     Cachesubf *subf;
     Cachefont **sub;    /* as read from file */
} Font;
```

The `height`, `ascent`, and `ldepth` fields of Font are described in *graphics*(2). `Sub` contains `nsub` pointers to `Cachefonts`. A `Cachefont` connects runes `min` through `max`, inclusive, to the subfont with file name `name`; it corresponds to a line of the file describing the font.

The image for rune *r* is found in position *r*–`min` of the subfont.

For each font, the library, with support from the graphics server, maintains a cache of subfonts and a cache of recently used character images. The `subf` and `cache` fields are used by the library to maintain these caches. The `width` of a font is the maximum of the horizontal extents of the characters in the cache. *String* draws a string by loading the cache and emitting a sequence of cache indices to draw. *Cachechars* guarantees the images for the characters pointed to by **s* are in the cache of *f.* It calls *loadchar* to put missing characters into the cache. *Cachechars* translates the character string into a set of cache indices which it loads into the array *c,* up to a maximum of *n* indices or the length of the string. *Cachechars* returns in *c* the number of cache indices emitted, updates **s* to point to the next character to be processed, and sets **widp* to the total width of the characters processed. *Cachechars* may return before the end of the string if it cannot proceed without destroying active data in the caches. It can return zero if it is unable to make progress because it is unable to resize the caches.

Loadchar loads a character image into the character cache. First, if necessary, it loads the subfont containing the character. Then it tells the graphics server to copy the character into position *h* in the character cache. If the current font `width` is smaller than the horizontal extent of the character being loaded, *loadfont* clears the cache and resets it to accept characters with the bigger width, unless *noclr* is set, in which case it just returns −1. If the character does not exist in the font at all, *loadfont* returns 0; if it is unable to load the character without destroying cached information, it returns −1.

The `age` fields record when subfonts and characters have been used. The font `age` is increased every time the font is used (*agefont* does this). A character or subfont `age` is set to the font age. Thus, characters or subfonts with small ages are the best candidates for replacement when the cache is full.

SEE ALSO

graphics(2), *balloc*(2), *bitblt*(2), *subfalloc*(2), *bitmap*(6), *font*(6)

DIAGNOSTICS

All of the functions use the graphics error function (see *graphics*(2)).

NAME

chdir − change working directory

SYNOPSIS

```
int chdir(char *dirname)
```

DESCRIPTION

Chdir changes the working directory of the invoking process to *dirname*. The working directory is the starting point for evaluating file names that do not begin with `/` or `#`, as explained in *intro*(2). When Plan 9 boots, the initial process has `/` for its working directory.

SEE ALSO

intro(2)

DIAGNOSTICS

Sets *errstr.*

NAME

cputime, times − cpu time in this process and children

SYNOPSIS

```
int    times(long t[4])

double cputime(void)
```

DESCRIPTION

Times fills in the array *t* with the number of milliseconds spent in user code, system calls, child processes in user code, and child processes in system calls. *Cputime* returns the sum of those same times, converted to seconds. *Times* returns the real time, in milliseconds used by the process so far.

These functions read `/dev/cputime`, opening that file when *time* is first called.

SEE ALSO

cons(3)

NAME

ctime, localtime, gmtime, asctime, timezone − convert date and time to ASCII

SYNOPSIS

```
char* ctime(long clock)

Tm*   localtime(long clock)

Tm*   gmtime(long clock)

char* asctime(Tm *tm)
```

DESCRIPTION

Ctime converts a time *clock* such as returned by *time*(2) into ASCII and returns a pointer to a 30-byte string in the following form. All the fields have constant width.

```
Wed Aug  5 01:07:47 EST 1973\n\0
```

Localtime and *gmtime* return pointers to structures containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT. *Asctime* converts a broken-down time to ASCII and returns a pointer to a 30-character string.

```
typedef
struct {
     int  sec;        /* seconds (range 0..59) */
     int  min;        /* minutes (0..59) */
     int  hour;       /* hours (0..23) */
     int  mday;       /* day of the month (1..31) */
     int  mon;        /* month of the year (0..11) */
     int  year;       /* year A.D. − 1900 */
     int  wday;       /* day of week (0..6, Sunday = 0) */
     int  yday;       /* day of year (0..365) */
     char zone[4];    /* time zone name */
} Tm;
```

When local time is first requested, the program consults the `timezone` environment variable to determine the time zone and converts accordingly. (This variable is set at system boot time by *init*(8).) The `timezone` variable contains the normal time zone name and its difference from GMT in seconds followed by an alternate (daylight) time zone name and its difference followed by a newline. The remainder is a list of pairs of times (seconds past the start of 1970, in the first time zone) when the alternate time zone applies. For example:

```
EST -18000 EDT -14400
 9943200 25664400 41392800 57718800 ...
```

Greenwich Mean Time is represented by

```
GMT 0
```

SEE ALSO

time(2), *init*(8)

BUGS

The return values point to static data whose content is overwritten by each call.
Daylight Savings Time is ‘‘normal’’ in the Southern hemisphere.
These routines are not equipped to handle non-ASCII text.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii, toascii, _toupper, _tolower, toupper, tolower − ASCII character classification

SYNOPSIS

```
#include <ctype.h>                          isprint(c)

isalpha(c)                                  isgraph(c)

isupper(c)                                  iscntrl(c)

islower(c)                                  isascii(c)

isdigit(c)                                  _toupper(c)

isxdigit(c)                                 _tolower(c)

isalnum(c)                                  toupper(c)

isspace(c)                                  tolower(c)

ispunct(c)                                  toascii(c)
```

DESCRIPTION

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* is true and on the single non-ASCII value `EOF`; *fopen*(2).

isalpha	*c* is a letter, a−z or A−Z
isupper	*c* is an upper case letter, A−Z
islower	*c* is a lower case letter, a−z
isdigit	*c* is a digit, 0−9
isxdigit	*c* is a hexadecimal digit, 0−9 or a−f or A−F
isalnum	*c* is an alphanumeric character, a−z or A−Z or 0−9
isspace	*c* is a space, horizontal tab, newline, vertical tab, formfeed, or carriage return (0x20, 0x9, 0xA, 0xB, 0xC, 0xD)
ispunct	*c* is a punctuation character (one of `!"#$%&'()*+,-./:;<=>?@[\]^_`‘`{|}~`)
isprint	*c* is a printing character, 0x20 (space) through 0x7E (tilde)
isgraph	*c* is a visible printing character, 0x21 (exclamation) through 0x7E (tilde)
iscntrl	*c* is a delete character, 0x7F, or ordinary control character, 0x0 through 0x1F
isascii	*c* is an ASCII character, 0x0 through 0x7F

Toascii is not a classification macro; it converts its argument to ASCII range by *and*ing with 0x7F.

If *c* is an upper case letter, *tolower* returns the lower case version of the character; otherwise it returns the original character. *Toupper* is similar, returning the upper case version of a character or the original character. *Tolower* and *toupper* are functions; *_tolower* and *_toupper* are corresponding macros which should only be used when it is known that the argument is upper case or lower case, respectively.

BUGS

These macros are ASCII-centric.

NAME

dial, hangup, announce, listen, accept, reject, netmkaddr − make and break network connections

SYNOPSIS

```
int   dial(char *addr, char *local, char *dir, int *cfdp)

int   hangup(int ctl)

int   announce(char *addr, char *dir)

int   listen(char *dir, char *newdir)

int   accept(int ctl, char *dir)

int   reject(int ctl, char *dir, char *cause)

char* netmkaddr(char *addr, char *defnet, char *defservice)
```

DESCRIPTION

For these routines, *addr* is a network address of the form *network* ! *netaddr* ! *service, network* ! *netaddr,* or simply *netaddr. Network* is any directory listed in `/net` or the special token, `net`. `Net` is a free variable that stands for any network in common between the source and the host *netaddr*. *Netaddr* can be a host name, a domain name, a network address, or a meta-name of the form `$`*attribute.* `$`*attribute* is replaced by *value* from the value-attribute pair *attribute*`=`*value* most closely associated with the source host in the network data base (see *ndb*(6)).

If a connection attempt is successful and *dir* is non-zero, the path name of a *line directory* that has files for accessing the connection is copied into *dir*. The path name is guaranteed to be less than 40 bytes long. One line directory exists for each possible connection. The `data` file in the line directory should be used to communicate with the destination. The `ctl` file in the line directory can be used to send commands to the line. See *dk*(3) and *ip*(3) for messages that can be written to the `ctl` file. The last close of the `data` or `ctl` file will close the connection.

Dial makes a call to destination *addr* on a multiplexed network. If the network in *addr* is `net`, *dial* will try in succession all networks in common between source and destination until a call succeeds. It returns a file descriptor open for reading and writing the `data` file in the line directory. The `addr` file in the line directory contains the address called. If the network allows the local address to be set, as is the case with UDP and TCP port numbers, and *local* is non-zero, the local address will be set to *local*. If *cfdp* is non-zero, `*`*cfdp* is set to a file descriptor open for reading and writing the control file.

Hangup is a means of forcing a connection to hang up without closing the `ctl` and `data` files.

Announce and *listen* are the complements of *dial. Announce* establishes a network name to which calls can be made. Like *dial*, *announce* returns an open `ctl` file. The *listen* routine takes as its first argument the *dir* of a previous *announce.* When a call is received, *listen* returns an open `ctl` file for the line the call was received on. It sets *newdir* to the path name of the new line directory. *Accept* accepts a call received by *listen*, while *reject* refuses the call because of *cause*. *Accept* returns a file descriptor for the data file opened `ORDWR .`

Netmkaddr make an address suitable for dialing or announcing. It takes an address along with a default network and service to use if they are not specified in the address. It returns a pointer to static data holding the actual address to use.

EXAMPLES

Make a call and return an open file descriptor to use for communications:

```
int callkremvax(void)
{
      return dial("kremvax", 0, 0, 0);
}
```

Call the local authentication server:

```
int dialauth(char *service)
{
      return dial(netmkaddr("$auth", 0, service), 0, 0, 0);
}
```

Announce as `kremvax` on Datakit and loop forever receiving calls and echoing back to the caller anything sent:

```
int
bekremvax(void)
{
      int dfd, acfd, lcfd;
      char adir[40], ldir[40];
      int n;
      char buf[256];

      afd = announce("dk!kremvax", adir);
      if(afd < 0)
            return -1;

      for(;;){
            /* listen for a call */
            lcfd = listen(adir, ldir);
            if(lcfd < 0)
                  return -1;

            /* fork a process to echo */
            switch(fork()){
            case -1:
                  perror("forking");
                  close(lcfd);
                  break;
            case 0:
                  /* accept the call and open the data file */
                  dfd = accept(lcfd, ldir);
                  if(dfd < 0)
                        return -1;

                  /* echo until EOF */
                  while((n = read(dfd, buf, sizeof(buf))) > 0)
                        write(dfd, buf, n);
                  exits(0);
            default:
                  close(lcfd);
                  break;
            }

      }
}
```

SEE ALSO

auth(2), *dk*(3), *ip*(3), *stream*(3), *ndb*(8)

DIAGNOSTICS

Dial, announce, and *listen* return −1 if they fail. *Hangup* returns nonzero if it fails.

NAME

dirread − read directory

SYNOPSIS

`int dirread(int fd, Dir *buf, long nbytes)`

DESCRIPTION

The data returned by a *read*(2) on a directory is a set of complete directory entries in a machine-independent format, exactly equivalent to the result of a *stat*(2) on each file or subdirectory in the directory. *Dirread* decodes the directory entries into a machine-dependent form. It reads from *fd* and unpacks the data into `Dir` structures in *buf* (see *stat*(2) for the layout of a `Dir`). *Nbytes* is the size of `buf`; it should be a multiple of `sizeof(Dir)`. Directory entries have length `DIRLEN` (defined in `<libc.h>`) in machine-independent form. A successful *read* of a directory always returns a multiple of `DIRLEN`; *dirread* always returns a multiple of `sizeof(Dir)`.

Dirread returns the number of bytes filled in `buf`; the number returned may be less than the number requested. The file offset is advanced by the number of bytes actually read.

SEE ALSO

intro(2), *open*(2), *read*(2)

DIAGNOSTICS

Sets *errstr.*

NAME

dup − duplicate an open file descriptor

SYNOPSIS

```
int dup(int oldfd, int newfd)
```

DESCRIPTION

Given a file descriptor, *oldfd*, referring to an open file, *dup* returns a new file descriptor referring to the same file. If *newfd* is in the range of legal file descriptors *dup* will use that for the new file descriptor (closing any old file associated with *newfd);* if *newfd* is −1 the system chooses the lowest available file descriptor.

SEE ALSO

intro(2), *dup*(3)

DIAGNOSTICS

Sets *errstr.*

NAME

encrypt, decrypt, netcrypt − DES encryption

SYNOPSIS

```
int   encrypt(void *key, void *data, int len)

int   decrypt(void *key, void *data, int len)

int   netcrypt(void *key, void *data)
```

DESCRIPTION

Encrypt and *decrypt* perform DES encryption and decryption. *Key* is an array of `DESKEYLEN` (defined as 7 in `<libc.h>`) bytes containing the encryption key. *Data* is an array of *len* bytes; it must be at least 8 bytes long. The bytes are encrypted or decrypted in place.

The DES algorithm encrypts an individual 8 byte block of data. *Encrypt* uses the following method to encrypt data longer than 8 bytes. The first 8 bytes are encrypted as usual. The last byte of the encrypted result is prefixed to the next 7 unencrypted bytes to make the next 8 bytes to encrypt. This is repeated until fewer than 7 bytes remain unencrypted. Any remaining unencrypted bytes are encrypted with enough of the preceding encrypted bytes to make a full 8 byte block. *Decrypt* uses the inverse algorithm.

Netcrypt performs the same encryption as a Securenet Box. *Data* points to an ASCII string of decimal digits with numeric value between 0 and 10000. These digits are copied into an 8 byte buffer with trailing binary zero fill and encrypted as one DES block. The first four bytes are each printed as two digit ASCII hexadecimal numbers, and the string is copied into data.

DIAGNOSTICS

These routines return 1 if the data was encrypted, and 0 if the encryption fails. *Encrypt* and *decrypt* fail if the data passed is less than 8 bytes long. *Netcrypt* can fail if it is passed invalid data.

SEE ALSO

securenet(8)

NAME

erf, erfc − error function

SYNOPSIS

```
double erf(double x)

double erfc(double x)
```

DESCRIPTION

These functions calculate the error function erf(x) and the complementary error function erfc(x). The error criterion for both *erf* and *erfc* is relative.

DIAGNOSTICS

There are no error returns.

NAME

errstr − description of last system call error

SYNOPSIS

```
int errstr(char *ans)
```

DESCRIPTION

When a system call fails it returns −1 and records a string describing the error in a per-process buffer. *Errstr* copies the contents of that buffer into the array *ans* and clears the buffer. *Ans* should contain at least `ERRLEN` characters (defined in `<libc.h>`).

The verb `r` in *print*(2) calls *errstr* and outputs the error string.

SEE ALSO

intro(2), *perror*(2)

NAME

event, einit, estart, etimer, eread, emouse, ekbd, ecanread, ecanmouse, ecankbd, ereshaped, getrect, menuhit, Event, Mouse, Menu − graphics events

SYNOPSIS

```
#include    <u.h>
#include    <libc.h>
#include    <libg.h>

void      einit(ulong keys)

ulong     event(Event *e)

Mouse     emouse(void)

int       ekbd(void)

int       ecanmouse(void)

int       ecankbd(void)

ulong     estart(ulong key, int fd, int n)

ulong     etimer(ulong key, int n)

ulong     eread(ulong keys, Event *e)

int       ecanread(ulong keys)

void      ereshaped(Rectangle r)

Rectangle getrect(int but, Mouse *m)

int       menuhit(int but, Mouse *m, Menu *menu)

enum{
          Emouse = 1,
          Ekeyboard = 2,
};
```

DESCRIPTION

These routines provide an interface to multiple sources of input. To use them, *einit* must be called. If the argument to *einit* has the `Emouse` and `Ekeyboard` bits set, the mouse and keyboard events will be enabled; in this case, *binit* (see *graphics*(2)) must have already been called. The user must provide a function called *ereshaped* to be called whenever the window in which the process is running has been reshaped; the argument will be the `Rectangle` for the new window shape, including the border.

As characters are typed on the keyboard, they are read by the event mechanism and put in a queue. *Ekbd* returns the next rune from the queue, blocking until the queue is non-empty. The characters are read in raw mode (see *cons*(3)), so they are available as soon as a complete rune is typed.

When the mouse moves or a mouse button is depressed or released, a new mouse event is queued by the event mechanism. *Emouse* returns the next mouse event from the queue, blocking until the queue is non-empty. *Emouse* returns a `Mouse` structure:

```
struct Mouse
{
     int   buttons;
     Point xy;
     ulong msec;
};
```

`Buttons&1` is set when the left mouse button is depressed, `buttons&2` when the middle button is depressed, and `buttons&4` when the right button is depressed. The current mouse position is always returned in `xy`. `Msec` is a time stamp in units of milliseconds.

Ecankbd and *ecanmouse* return non-zero when there are keyboard or mouse events available to be read.

Estart can be used to register additional file descriptors to scan for input. It takes as arguments the file descriptor to register, the maximum length of an event message on that descriptor, and a key to be used in accessing the event. The key must be a power of 2 and must not conflict with any previous keys. If a zero key is given, one will be allocated and returned. `Ekeyboard` and `Emouse` are the mouse and keyboard event keys.

Etimer starts a repeating timer with a period of *n* milliseconds. Only one timer can be started. Extra timer events are not queued and the timer channel has no associated data.

Eread waits for the next event specified by the mask `keys` of event keys submitted to `estart`. It fills in the appropriate field of the argument `Event` structure, which looks like:

```
struct Event
{
     int   kbdc;
     Mouse mouse;
     int   n;
     uchar data[EMAXMSG];
};
```

`Data` is an array which is large enough to hold a 9P message. *Eread* returns the key for the event which was chosen. For example, if a mouse event was read, *Emouse* will be returned.

Event waits for the next event of any kind. The return is the same as for *eread*.

As described in *graphics*(2), the graphics functions are buffered. *Event*, *eread*, *emouse*, and *ekbd* all cause a buffer flush unless there is an event of the appropriate type already queued.

Getrect prompts the user to sweep a rectangle. It should be called with *m* holding the mouse event that triggered the *getrect* (or, if none, a `Mouse` with `buttons` set to 7). It changes to the sweep cursor, waits for the buttons all to be released, and then waits for button number *but* to be depressed, marking the initial corner. If another button is depressed instead, *getrect* returns a rectangle with zero for both corners, after waiting for all the buttons to be released. Otherwise, *getrect* continually draws the swept rectangle until the button is released again, and returns the swept rectangle. The mouse structure pointed to by *m* will contain the final mouse event.

Menuhit displays a menu and returns a selected menu item number. It should be called with *m* holding the mouse event that triggered the *menuhit*; it will call *emouse* to update it. A `Menu` is a structure:

```
struct Menu
{
     char  **item;
     char  *(*gen)(int);
     int   lasthit;
};
```

If `item` is nonzero, it should be a null-terminated array of the character strings to be displayed as menu items. Otherwise, `gen` should be a function that, given an item number, returns the character string for that item, or zero if the number is past the end of the list. Items are numbered starting at zero. *Menuhit* waits until *but* is released, and then returns the number of the selection, or −1 for no selection. The *m* argument is filled in with the final mouse event.

SEE ALSO

8½(1), *graphics*(2), *cons*(3), *bit*(3)

NAME

exec, execl − execute a file

SYNOPSIS

`int exec(char *name, char* argv[])`

`int execl(char *name, ...)`

DESCRIPTION

Exec and *execl* overlay the calling process with the named file, then transfer to the entry point of the image of the file.

Name points to the name of the file to be executed; it must not be a directory, and the permissions must allow the current user to execute it (see *stat*(2)). It should also be a valid binary image, as defined in the *a.out*(6) for the current machine architecture, or a shell script (see *rc*(1)). The first line of a shell script must begin with `#!` followed by the name of the program to interpret the file and any initial arguments to that program, for example

```
#!/bin/rc
ls | mc
```

When a C program is executed, it is called as follows:

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

Argv is a copy of the array of argument pointers passed to *exec*; that array must end in a null pointer, and *argc* is the number of elements before the null pointer. By convention, the first argument should be the name of the program to be executed. *Execl* is like *exec* except that *argv* will be an array of the parameters that follow *name* in the call. The last argument to *execl* must be a null pointer.

For a file beginning `#!`, the arguments passed to the program (`/bin/rc` in the example above) will be the name of the file being executed, any arguments on the `#!` line, the name of the file again, and finally the second and subsequent arguments given to the original *exec* call. The result honors the two conventions of a program accepting as argument a file to be interpreted and `argv[0]` naming the file being executed.

Most attributes of the calling process are carried into the result; in particular, files remain open across *exec* (except those opened with `OCEXEC` OR'd into the open mode; see *open*(2)); and the working directory and environment (see *env*(3)) remain the same However, a newly *exec'ed* process has no notification handler (see *notify*(2)).

When the new program begins, the global cell `_clock` is set to the address of a cell that keeps approximate time expended by the process at user level. The time is measured in milliseconds but is updated at a system-dependent lower rate. This clock is typically used by the profiler but is available to all programs.

The above conventions apply to C programs; the raw system interface to the new image is as follows: the word pointed to by the stack pointer is `argc`; the words beyond that are the zeroth and subsequent elements of `argv`, followed by a terminating null pointer; and the return register (e.g. `R0` on the 68020) contains the address of the clock.

SEE ALSO

intro(2), *stat*(2)

DIAGNOSTICS

If these functions fail, they return and set *errstr.* There can be no return from a successful *exec* or *execl*; the calling image is lost.

NAME

exits, atexit, atexitdont − terminate process, process cleanup

SYNOPSIS

```
void  _exits(char *msg)
void  exits(char *msg)

int   atexit(void(*)(void))

void  atexitdont(void(*)(void))
```

DESCRIPTION

Exits is the conventional way to terminate a process. *_Exits* is the underlying system call. They can never return.

Msg conventionally includes a brief (maximum length `ERRLEN`) explanation of the reason for exiting, or a null pointer or empty string to indicate normal termination. The string is passed to the parent process, prefixed by the name and process id of the exiting process, when the parent does a *wait*(2).

Before calling *_exits* with *msg* as an argument, *exits* calls in reverse order all the functions recorded by *atexit*.

Atexit records *fn* as a function to be called by *exits*. It returns zero if it failed, nonzero otherwise. A typical use is to register a cleanup routine for an I/O package.

Calling *atexit* twice (or more) with the same function argument causes *exits* to invoke the function twice (or more).

There is a limit to the number of exit functions that will be recorded; *atexit* returns 0 if that limit has been reached.

Atexitdont cancels a previous registration of an exit function, which is useful after a *fork*(2) to avoid conflicting calls of an exit function

SEE ALSO

fork(2), *wait*(2)

NAME

exp, log, log10, pow, pow10, sqrt − exponential, logarithm, power, square root

SYNOPSIS

```
double exp(double x)

double log(double x)

double log10(double x)

double pow(double x, double y)

double pow10(int n)

double sqrt(double x)
```

DESCRIPTION

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

Pow returns x^y and *pow10* returns 10^n as a double.

Sqrt returns the square root of *x*.

SEE ALSO

hypot(2), *sinh*(2), *intro*(2)

NAME

fcall, convS2M, convD2M, convM2S, convM2D, getS, fcallconv, dirconv, dirmodeconv − interface to Plan 9 File protocol

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <fcall.h>

int convS2M(Fcall *f, char *ap)

int convD2M(Dir *d, char *ap)

int convM2S(char *ap, Fcall *f, int n)

int convM2D(char *ap, Dir *d)

char *getS(int fd, char *ap, Fcall *f, long *lp)

int dirconv(void *o, int f1, int f2, int f3, int chr)

int fcallconv(void *o, int f1, int f2, int f3, int chr)

int dirmodeconv(void *o, int f2, int f2, int f3, int chr)
```

DESCRIPTION

These routines convert messages in the machine-independent format of the Plan 9 file protocol, 9P, to and from a more convenient form, an `Fcall` structure:

```
typedef
struct Fcall {
    char  type;
    short fid;
    short tag;
    union {
          struct {
              ushort                oldtag;/* Tflush */
              Qid   qid;            /* Rattach, Rwalk, Ropen, Rcreate */
          };
          struct {
              char  uname[NAMELEN]; /* Tauth, Tattach */
              char  aname[NAMELEN]; /* Tattach */
              char  auth[NAMELEN];  /* Tattach */
              char  chal[8+NAMELEN];/* Tauth, Rauth */
          };
          struct {
              char  ename[ERRLEN];  /* Rerror */
          };
          struct {
              long  perm;           /* Tcreate */
              short newfid;         /* Tclone, Tclwalk */
              char  name[NAMELEN];  /* Twalk, Tclwalk, Tcreate */
              char  mode;           /* Tcreate, Topen */
          };
          struct {
              long  offset;         /* Tread, Twrite */
              long  count;          /* Tread, Twrite, Rread */
              char  *data;          /* Twrite, Rread */
          };
          struct {
```

```
                char  stat[DIRLEN];   /* Twstat, Rstat */
            };
        };
} Fcall;
```

This structure is defined in `<fcall.h>`. See section 5 for a full description of 9P messages and their encoding. For all message types, the `type` field of an `Fcall` holds one of `Tnop`, `Rnop`, `Tsession`, `Rsession`, etc. (defined in an enumerated type in `<fcall.h>`). `Fid` is used by most messages, and `tag` is used by all messages. The other fields are used selectively by the message types given in comments.

ConvM2S takes a 9P message at *ap* of length *n*, and uses it to fill in `Fcall` structure *f*. If the passed message including any data for `Twrite` and `Rread` messages is formatted properly, the return value is *n*; otherwise it is 0. For `Twrite` and `Tread` messages, `data` is set to a pointer into the argument message, not a copy.

ConvS2M does the reverse conversion, turning *f* into a message starting at *ap*. The length of the resulting message is returned. For `Twrite` and `Rread` messages, `count` bytes starting at `data` are copied into the message.

The constant `MAXMSG` is the length of the longest message, excluding data; `MAXFDATA` (8192) is the maximum `count` in a read or write message. Thus messages are guaranteed to be shorter than `MAXMSG+MAXFDATA` bytes long.

Another structure is `Dir`, used by the routines described in *stat*(2). *ConvM2D* converts the machine-independent form starting at *ap* into *d* and returns the length of the encoding. *ConvD2M* does the reverse translation, also returning the length of the encoding.

GetS reads a message from file descriptor *fd* into *ap* and converts the message using *convM2S* into the `Fcall` structure *f*. The *lp* argument must point to a `long` holding the size of the *ap* buffer. It is somewhat resilient to transient read errors. If *convM2S* succeeds, its return value is stored in **lp*, and *getS* returns zero. Otherwise *getS* returns a string identifying the error.

Dirconv, *fcallconv*, and *dirmodeconv* are formatting routines, suitable for *fmtinstall* (see *print*(2)). They convert `Dir*`, `Fcall*`, and `long` values into string representations of the directory buffer, `Fcall` buffer, or file mode value. *Fcallconv* assumes that *dirconv* has been installed with format letter `D`.

SEE ALSO

intro(2), *stat*(2), *intro*(5)

DIAGNOSTICS

GetS sets *errstr.*

BUGS

The offset and directory length fields have 8 bytes in the protocol, but these routines assume they fit into a long.

ConvS2M should check for counts exceeding `MAXFMSG`.

NAME

fgetc, getc, getchar, fputc, putc, putchar, ungetc, fgets, gets, fputs, puts, fread, fwrite − stdio input and output

SYNOPSIS

```
#include <stdio.h>

int  fgetc(FILE *f)

int  getc(FILE *f)

int  getchar(void)

int  fputc(int c, FILE *f)

int  putc(int c, FILE *f)

int  putchar(int c)

int  ungetc(int c, FILE *f)

char *fgets(char *s, int n, FILE *f)

char *gets(char *s)

int  fputs(char *s, FILE *f)

int  puts(char *s)

long fread(void *ptr, long itemsize, long nitems, FILE *stream)

long fwrite(void *ptr, long itemsize, long nitems, FILE *stream)
```

DESCRIPTION

The functions described here work on open stdio streams (see *fopen*).

Fgetc returns as an `int` the next `unsigned char` from input stream *f*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and *fgetc* returns `EOF`. If a read error occurs, the error indicator for the stream is set and *fgetc* returns `EOF`. *Getc* is like *fgetc* except that it is implemented as a macro. *Getchar* is like *getc* except that it always reads from `stdin`.

Ungetc pushes character *c* back onto the input stream `f`. The pushed-back character will be returned by subsequent reads in the reverse order of their pushing. A successful intervening *fseek*, *fsetpos*, or *rewind* on *f* discards any pushed-back characters for *f*. One character of pushback is guaranteed. *Ungetc* returns the character pushed back (converted to `unsigned char`), or `EOF` if the operation fails. A successful call to *ungetc* clears the end-of-file indicator for the stream. The file position indicator for the stream after reading or discarding all pushed-back characters is the same as it was before the characters were pushed back.

Fputc writes character *c* (converted to `unsigned char`) to output stream *f* at the position indicated by the position indicator for the stream and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. *Fputc* returns the character written or `EOF` if there was a write error. *Putc* is like *fputc* but is implemented as a macro. *Putchar* is like *putc* except that it always writes to `stdout`.

All other input takes place as if characters were read by successive calls to *fgetc* and all other output takes place as if characters were written by successive calls to *fputc*.

Fgets reads up to and including the next newline, but not past end-of-file or more than n-1 characters, from stream *f* into array *s*. A null character is written immediately after the last character read into the array (if any characters are read at all). *Fgets* returns *s* if successful, otherwise a null pointer. *Gets* is similar to *fgets* except that it always reads from `stdin` and it discards the terminating newline, if any. *Gets* does not check for overflow of the receiving array, so its use is deprecated.

Fputs writes the string *s* to stream *f*, returning `EOF` if a write error occurred, otherwise a nonnegative value. The terminating null character is not written. *Puts* is the same, writing to `stdout`.

Fread reads from the named input *stream* at most *nitems* of data of the type of **ptr* into a block beginning at *ptr.* It returns the number of items actually read.

Fwrite appends to the named output *stream* at most *nitems* of data of the type of **ptr* from a block beginning at *ptr.* It returns the number of items actually written.

SEE ALSO

read(2), *fopen*(2), *bio*(2)

NAME

fabs, fmod, floor, ceil − absolute value, remainder, floor, ceiling functions

SYNOPSIS

```
double floor(double x)

double ceil(double x)

double fabs(double x)

double fmod(double x, double y)
```

DESCRIPTION

Fabs returns the absolute value $|x|$.

Floor returns the largest integer not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns x if y is zero, otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |x|$.

SEE ALSO

abs(2), *frexp*(2)

NAME

fopen, freopen, fdopen, fileno, fclose, sopenr, sopenw, sclose, fflush, setvbuf, setbuf, fgetpos, ftell, fsetpos, fseek, rewind, feof, ferror, clearerr − standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(char *filename, char *mode)

FILE *freopen(char *filename, char *mode, FILE *f)

FILE *fdopen(int fd, char *mode)

int  fileno(FILE *f)

FILE *sopenr(char *s)

FILE *sopenw(void)

char *sclose(FILE *f)

int  fclose(FILE *f)

int  fflush(FILE *f)

int  setvbuf(FILE *f, char *buf, int type, long size)

void setbuf(FILE *f, char *buf)

int  fgetpos(FILE *f, long *pos)

long ftell(FILE *f)

int  fsetpos(FILE *f, long *pos)

int  fseek(FILE *f, long offset, int whence)

void rewind(FILE *f)

int  feof(FILE *f)

int  ferror(FILE *f)

void clearerr(FILE *f)
```

DESCRIPTION

The functions described in Section 2S constitute the ANSI standard buffered I/O package with extensions.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type `FILE`. *Fopen*(2) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

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

A constant pointer `NULL` designates no stream at all.

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, or `NULL` if the open fails. *Mode* is a character string having one of the following values:

`"r"`	open for reading
`"w"`	truncate to zero length or create for writing
`"a"`	append; open or create for writing at end of file
`"r+"`	open for update (reading and writing)
`"w+"`	truncate to zero length or create for update
`"a+"`	append; open or create for update at end of file

In addition, each of the above strings can have a `b` somewhere after the first character, meaning 'binary file', but this implementation makes no distinction between binary and text files.

Fclose causes the stream pointed to by *f* to be flushed (see below) and does a *close* (see *open*(2)) on the associated file. It frees any automatically allocated buffer. *Fclose* is called automatically on *exits*(2) for all open streams.

Freopen is like open except that it reuses stream pointer *f*. *Freopen* first attempts to close any file associated with *f*; it ignores any errors in that close.

Fdopen associates a stream with an open Plan 9 file descriptor.

Fileno returns the number of the Plan 9 file descriptor associated with the stream.

Sopenr associates a read-only stream with a null-terminated string.

Sopenw opens a stream for writing. No file descriptor is associated with the stream; instead, all output is written to the stream buffer.

Sclose closes a stream opened with *sopenr* or *sopenw*. It returns a pointer to the 0 terminated buffer associated with the stream.

By default, output to a stream is fully buffered: it is accumulated in a buffer until the buffer is full, and then *write* (see *read*(2)) is used to write the buffer. An exception is standard error, which is line buffered: output is accumulated in a buffer until a newline is written. Input is also fully buffered by default; this means that *read*(2) is used to fill a buffer as much as it can, and then characters are taken from that buffer until it empties. *Setvbuf* changes the buffering method for file *f* according to *type:* either `_IOFBF` for fully buffered, `_IOLBF` for line buffered, or `_IONBF` for unbuffered (each character causes a *read* or *write).* If *buf* is supplied, it is used as the buffer and *size* should be its size; If *buf* is zero, a buffer of the given size is allocated (except for the unbuffered case) using *malloc*(2).

Setbuf is an older method for changing buffering. If *buf* is supplied, it changes to fully buffered with the given buffer, which should be of size `BUFSIZ` (defined in `stdio.h`). If *buf* is zero, the buffering method changes to unbuffered.

Fflush flushes the buffer of output stream *f*, delivering any unwritten buffered data to the host file.

There is a *file position indicator* associated with each stream. It starts out pointing at the first character (unless the file is opened with append mode, in which case the indicator is always ignored). The file position indicator is maintained by the reading and writing functions described in *fgetc*(2).

Fgetpos stores the current value of the file position indicator for stream *f* in the object pointed to by *pos*. It returns zero on success, nonzero otherwise. *Ftell* returns the current value of the file position indicator. The file position indicator is to be used only as an argument to *fseek.*

Fsetpos sets the file position indicator for stream *f* to the value of the object pointed to by *pos*, which shall be a value returned by an earlier call to *fgetpos* on the same stream. It returns zero on success, nonzero otherwise. *Fseek* obtains a new position, measured in characters from the beginning of the file, by adding *offset* to the position specified by *whence*: the beginning of the file if *whence* is `SEEK_SET`; the current value of the file position indicator for `SEEK_CUR`; and the end-of-file for `SEEK_END`. *Rewind* sets the file position indicator to the beginning of the file.

An integer constant `EOF` is returned upon end of file or error by integer-valued functions that deal with streams. *Feof* returns non-zero if and only if *f* is at its end of file.

Ferror returns non-zero if and only if *f* is in the error state. It can get into the error state if a system call failed on the associated file or a memory allocation failed. *Clearerr* takes a stream out of the error state.

SEE ALSO

fprintf(2), *fscanf*(2), *fgetc*(2)
open(2), *read*(2)

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.

BUGS

Buffering of output can prevent output data from being seen until long after it is computed − perhaps never, as when an abort occurs between buffer filling and flushing.
Buffering of input can cause a process to consume more input than it actually uses. This can cause trouble across *exec*(2).
Buffering may delay the receipt of a write error until a subsequent *stdio* writing, seeking, or file-closing call.
ANSI says that a file can be fully buffered only if the file is not attached to an interactive device. In Plan 9 all are fully buffered except standard error.

Fdopen, *fileno*, *sopenr*, *sopenw*, and *sclose* are not ANSI stdio functions.

Stdio offers no support for runes or UTF characters. Unless external compatibility is necessary, use *bio*(2), which supports UTF and is smaller, faster, and simpler than stdio.

NAME

fork, rfork − manipulate process resources

SYNOPSIS

```
int fork(void)

int rfork(int flags)
```

DESCRIPTION

Forking is the only way new processes are created. The *flags* argument to *rfork* selects which resources of the invoking process (parent) are shared by the new process (child) or initialized to their default values. The resources include the file name space, the open file descriptor table (which, when shared, permits processes to open and close files for other processes), the set of environment variables (see *env*(3)), the note group (the set of processes that receive notes written to a member's `notepg` file; see *proc*(3)), and open files. *Flags* is the logical OR of some subset of

`RFPROC`
: If set a new process is created; otherwise changes affect the current process.

`RFNAMEG`
: If set, the new process inherits a copy of the parent's name space; otherwise the new process shares the parent's name space. Is mutually exclusive with `RFCNAMEG`.

`RFNOWAIT`
: If set, the child process will be disassociated from the parent. Upon exit the child will leave no `Waitmsg` (see *wait*(2)) for the parent to collect.

`RFCNAMEG`
: If set, the new process starts with a clean name space. A new name space must be built from a mount of an open file descriptor. Is mutually exclusive with `RFNAMEG`.

`RFENVG`
: If set, the environment variables are copied; otherwise the two processes share environment variables. Is mutually exclusive with `RFCENVG`.

`RFCENVG`
: If set, the new process starts with an empty environment. Is mutually exclusive with `RFENVG`.

`RFNOTEG`
: Each process is a member of a group of processes that all receive notes when a note is written to any of their `notepg` files (see *proc*(3)). The group of a new process is by default the same as its parent, but if `RFNOTEG` is set (regardless of `RFPROC`), the process becomes the first in a new group, isolated from previous processes.

`RFFDG` If set, the invoker's file descriptor table (see *intro*(2)) is copied; otherwise the two processes share a single table.

`RFCFDG`
: If set, the new process starts with a clean file descriptor table. Is mutually exclusive with `RFCFDG`.

`RFMEM` If set, the kernel will mark segments of type `data` and `bss` as shared. The child will then inherit all the shared segments the parent process owns. Other segment types will be unaffected. Subsequent forks by the parent will then propagate the shared data and bss between children. The stack segment is always split. May be set only with `RFPROC`.

File descriptors in a shared file descriptor table are kept open until either they are explicitly closed or all processes sharing the table exit.

If `RFPROC` is set, the value returned in the parent process is the process id of the child process; the value returned in the child is zero. Without `RFPROC`, the return value is zero. Process ids range from 1 to the maximum integer (`int`) value. *Rfork* will sleep, if necessary, until required process resources are available.

Fork is just a call of `rfork(RFFDG|RFPROC)`.

SEE ALSO

intro(2), *proc*(3),

DIAGNOSTICS

These functions set *errstr.*

NAME

fprintf, printf, sprintf, vfprintf, vprintf, vsprintf − print formatted output

SYNOPSIS

```
#include <stdio.h>

int fprintf (FILE *f, char *format, ...);

int printf(char *format, ...);

int sprintf (char *s, char *format, ...);

int vfprintf (FILE *f, char *format, char *args);

int vprintf(char *format, char *args);

int vsprintf (char *s, char *format, char *args);
```

DESCRIPTION

Fprintf places output on the named output stream *f* (see *fopen*(2)). *Printf* places output on the standard output stream *stdout*. *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. *Vfprintf*, *vprintf*, and *vsprintf* are the same, except the *args* argument is a pointer to an argument in an argument list of the calling function, and the effect is as if the calling function's argument list from that point on is passed to the *printf* routines.

Each function returns the number of characters transmitted (not including the `\0` in the case of *sprintf*), or a negative value if an output error was encountered.

These functions convert, format, and print their trailing arguments under control of a *format* string. The *format* contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more arguments. The results are undefined if there are arguments of the wrong type or too few arguments for the format. If the format is exhausted while arguments remain, the excess are 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 with spaces on the left (or right, if the left adjustment, described later, has been given) to the field width.
>
> An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `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` conversions, or the maximum number of characters to be written from a string in `s` conversion. The precision takes the form of a period (`.`) followed by an optional decimal integer; if the integer is omitted, it is treated as zero.
>
> An optional `h` specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion specifier applies to a `short int` or `unsigned short` argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to `short` or `unsigned short` before printing); an optional `h` specifying that a following `n` conversion specifier applies to a pointer to a `short` argument; an optional `l` (ell) specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to a `long` or `unsigned long` argument; an optional `l` specifying that a following `n` conversion specifier applies to a pointer to a `long int` argument; or an optional `L` specifying that a following `e`, `E`, `f`, `g`, or `G` conversion specifier applies to a `long double` argument. If an `h`, `l`, or `L` appears with any other conversion specifier, the behavior is undefined.
>
> A character that indicates the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk (`*`) instead of a digit string. In this case, an `int` *arg* supplies the field width or precision. The arguments specifying field width or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a − flag followed by a positive field width. A negative precision is taken as if it were missing.

The flag characters and their meanings are:

−	The result of the conversion is left-justified within the field.
+	The result of a signed conversion always begins with a sign (+ or −).
blank	If the first character of a signed conversion is not a sign, or a signed conversion results in no characters, a blank is prefixed to the result. This implies that if the blank and + flags both appear, the blank flag is ignored.
`#`	The result is to be converted to an ''alternate form.'' 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 has `0x` or `0X` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result always contains 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 zeros are *not* be removed from the result as they normally are. For other conversions, the behavior is undefined.
`0`	For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad the field width; no space padding is performed. If the `0` and − flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. For other conversions, the behavior is undefined.

The conversion characters and their meanings are:

`d,o,u,x,X`	The integer *arg* is converted to signed decimal (`d` or `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`); 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 is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.
`f`	The `double` argument 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, it is taken as 6; if the precision is explicitly `0`, no decimal point appears.
`e,E`	The `double` argument is converted in the style [−]*d*`.`*ddd*`e`±*dd*, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, it is taken as 6; if the precision is zero, no decimal point appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.
`g,G`	The `double` argument is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it is taken as 1. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal point appears only if it is followed by a digit.
`c`	The `int` argument is converted to an `unsigned char`, and the resulting character is written.
`s`	The argument is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or 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 zero value for the argument yields undefined results.
`P`	The `void *` argument is printed in an implementation defined way (for Plan 9: the address as hexadecimal number).
`n`	The argument shall be a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to *fprintf*. No argument is converted.

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

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer cast to be a pointer to `void` using `%P` conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

SEE ALSO

fopen(2), *fscanf*(2), *print*(2)

BUGS

There is no way to print a wide character (rune).

NAME

frinit, frsetrects, frclear, frcharofpt, frptofchar, frinsert, frdelete, frselect, frselectp, frselectf, frgetmouse − frames of text

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>
#include <frame.h>

void  frinit(Frame *f, Rectangle r, Font *ft, Bitmap *b);

void  frsetrects(Frame *f, Rectangle r, Bitmap *b);

void  frclear(Frame *f);

ulong frcharofpt(Frame *f, Point pt);

Point frptofchar(Frame *f, ulong p);

void  frinsert(Frame *f, Rune *r0, Rune *r1, ulong p);

int   frdelete(Frame *f, ulong p0, ulong p1);

void  frselect(Frame *f, Mouse *m);

void  frselectp(Frame *f, Fcode fc);

void  frselectf(Frame *f, Point p0, Point p1, Fcode c);

extern void frgetmouse(void);
```

DESCRIPTION

This library supports *frames* of editable text in a single font on bitmap displays, such as in *sam*(1) and *8½*(1). Frames may hold any character except NUL (0). Long lines are folded and tabs are at fixed intervals.

The user-visible data structure, a `Frame`, is defined in `<frame.h>`:

```
typedef struct Frame Frame;
struct Frame
{
     Font      *font;          /* of chars in the frame */
     Bitmap    *b;             /* on which frame appears */
     Rectangle r;              /* in which text appears */
     Rectangle entire;         /* of full frame */
     Frbox     *box;
     ulong     p0, p1;         /* selection */
     short     left;           /* left edge of text */
     ushort    nbox, nalloc;
     ushort    maxtab;         /* max size of tab, in pixels */
     ushort    nchars;         /* # runes in frame */
     ushort    nlines;         /* # lines with text */
     ushort    maxlines;       /* total # lines in frame */
     ushort    lastlinefull;   /* last line fills frame */
     ushort    modified;       /* changed since frselect() */
};
```

`Frbox` is an internal type and is not used by the interface. `P0` and `p1` may be changed by the application provided the selection routines are called afterwards to maintain a consistent display. *Maxtab* determines the size of tab stops. *Frinit* sets it to 8 times the width of a `0` (zero) character in the font; it may be changed before any text is added to the frame. The other elements of the structure are maintained by the library and should not be modified directly.

The text within frames is not directly addressable; instead frames are designed to work alongside another structure that holds the text. The typical application is to display a section of a longer document such as a text file or terminal session. Usually the application will keep its own copy of the text in the window (probably as an array of `Runes`) and pass components of this text to the frame routines to display the visible portion. Only the text that is visible is held by the `Frame`; the application must check `maxlines`, `nlines`, and `lastlinefull` to determine, for example, whether new text needs to be appended at the end of the `Frame` after calling *frdelete* (q.v.).

There are no routines in the library to allocate `Frames`; instead the interface assumes that `Frames` will be components of larger structures. *Frinit* prepares the `Frame` *f* so characters drawn in it will appear in the single `Font` *ft.* It then calls `frsetrects` to initialize the geometry for the `Frame.` The `Bitmap` *b* is where the `Frame` is to be drawn; `Rectangle` *r* defines the limit of the portion of the `Bitmap` the text will occupy. The `Bitmap` pointer may be null, allowing the other routines to be called to maintain the associated data structure in, for example, an obscured window.

Frclear frees the internal structures associated with *f,* permitting another *frinit* or *frsetrects* on the `Frame`. If *f* is to be deallocated, the associated `Font` and `Bitmap` must be freed separately.

To reshape a `Frame`, use *frclear* and *frinit* and then *frinsert* (q.v.) to recreate the display. If a `Frame` is being moved but not reshaped, that is, if the shape of its containing rectangle is unchanged, it is sufficient to *bitblt*(2) the containing rectangle from the old to the new location and then call *frsetrects* to establish the new geometry. No redrawing is necessary.

`Frames` hold text as runes, not as bytes. *Frptofchar* returns the location of the upper left corner of the *p'th* rune in the `Frame` *f.* If *f* holds fewer than *p* runes, *frptofchar* returns the location of the upper right corner of the last character in *f. Frcharofpt* is the inverse: it returns the index of the closest rune whose image's upper left corner is up and to the left of *pt.*

Frinsert inserts into `Frame` *f* starting at rune index *p* the runes between *r0* and *r1.* If a NUL (0) character is inserted, chaos will ensue. Tabs and newlines are handled by the library, but all other characters, including control characters, are just displayed. For example, backspaces are printed; to erase a character, use *frdelete.*

Frdelete deletes from the `Frame` the text between *p0* and *p1*; *p1* points at the first rune beyond the deletion.

Frselect tracks the mouse to select a contiguous string of text in the `Frame`. When called, mouse button 1 should be depressed. It will return when the button is released and will set *f*`->p0` and *f*`->p1` to the selected range of text. *Frselectf* and *Frselectp* modify the display of the selected text. *Frselectf* highlights the text between *p0* and *p1* (which must have been returned by *frptofochar*) using `bitblt` in mode *c. Frselectp* is similar but highlights the text from *f*`->p0` to *f*`->p1`. Neither *frselectf* nor *frselectp* modifies *f*`->p0` or *f*`->p1`.

Upon return from *frinsert* or *frdelete,* the display will be consistent but *f*`->p0` and *f*`->p1` may not point to the desired selection. It may be necessary to adjust the selection and use *frselectf* or *frselectp* to fix the display.

Frgetmouse must be provided by the application; *frselect* calls it to get mouse updates. Each call to *frgetmouse* should update the `Mouse` structure pointed to by *frselect's* argument *m. Frgetmouse* should block until the mouse status has changed.

SEE ALSO

graphics(2), *bitblt*(2), *cachechars*(2).

NAME

frexp, ldexp, modf − split into mantissa and exponent

SYNOPSIS

```
double frexp(double value, int *eptr)

double ldexp(double value, int exp)

double modf(double value, double *iptr)
```

DESCRIPTION

Frexp returns the mantissa of *value* and stores the exponent indirectly through *eptr,* so that $value = frexp(value) \times 2^{*eptr}$

Ldexp returns the quantity $value \times 2^{exp}$.

Modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr.*

DIAGNOSTICS

Ldexp returns 0 for underflow and the appropriately signed infinity for overflow.

SEE ALSO

intro(2)

NAME

fscanf, scanf, sscanf, vfscanf − scan formatted input

SYNOPSIS

```
#include <stdio.h>

int fscanf (FILE *f, char *format, ...);

int scanf(char *format, ... );

int sscanf (char *s, char *format, ...);

int vfscanf (FILE *stream, char *format, char *args);
```

DESCRIPTION

Fscanf reads from the named input stream *f* (see *fopen*(2)) under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

Scanf and *sscanf* are the same, but they read from *stdin* and the character string *s*, respectively. *Vfscanf* is like *scanf*, except the *args* argument is a pointer to an argument in an argument list of the calling function and the effect is as if the calling function's argument list from that point on is passed to the scanf routines.

The format is composed of zero or more directives: one or more white-space characters; an ordinary character (not `%`); or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

> An optional assignment-suppressing character `*`.
>
> An optional decimal integer that specifies the maximum field width.
>
> An optional `h`, `l` (ell) or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` shall be preceded by `h` if the corresponding argument is a pointer to `short` rather than a pointer to `int`, or by `l` if it is a pointer to `long`. Similarly, the conversion specifiers `o`, `u`, and `x` shall be preceded by `h` if the corresponding argument is a pointer to `unsigned short` rather than a pointer to `unsigned`, or by `l` if it is a pointer to `unsigned long`. Finally, the conversion specifiers `e`, `f`, and `g` shall be preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. If an `h`, `l`, or `L` appears with any other conversion specifier, the behavior is undefined.
>
> A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

Fscanf executes each directive of the format in turn. If a directive fails, as detailed below, *fscanf* returns. Failures are described as input failures (due to the unavailability of input), or matching failures (due to inappropriate input).

A directive composed of white space is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next character from the stream. If if differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by *isspace*, see *ctype*(2)) are skipped, unless the specification includes a `[`, `c`, or `n` specifier.

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

`d`	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *strtol* (see *atof*(2)) function with 10 for the `base` argument. The corresponding argument shall be a pointer to `int`.
`i`	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *strtol* function with 0 for the `base` argument. The corresponding argument shall be a pointer to `int`.
`o`	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the *strtoul* (see *atof*(2)) function with 8 for the `base` argument. The corresponding argument shall be a pointer to `unsigned int`.
`u`	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *strtoul* function with 10 for the `base` argument. The corresponding argument shall be a pointer to `unsigned int`.
`x`	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the *strtoul* function with 16 for the `base` argument. The corresponding argument shall be a pointer to `unsigned int`.
`e,f,g`	Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the *strtod* (see *atof*(2)) function. The corresponding argument shall be a pointer to `float` .
`s`	Matches a sequence of non-white-space characters. The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating NUL (0) character, which will be added automatically.
`[`	Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating NUL character, which will be added automatically. The conversion specifier includes all subsequent characters in the *format* string, up to and including the matching right brace (`]`). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. As a special case, if the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification. If a − character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined (in Plan 9: the scanlist includes all characters in the ASCII range between the two characters on either side of the −).
`c`	Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No NUL character is added.
`P`	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%P` conversion of the *fprintf*(2) function. The corresponding

argument shall be a pointer to a pointer to `void`. The interpretation of the input item is implementation defined; however, for any input item other than a value converted earlier during the same program execution, the behavior of the `%P` conversion is undefined.

`n` No input is consumed. The corresponding argument shall be a pointer to integer into which is written the number of characters read from the input stream so far by this call to *fscanf*. Execution of a `%n` directive does not increment the assignment count returned at the completion of *fscanf*.

`%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `E`, `G`, and `X` are also valid and behave the same as, respectively, `e`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

The return value from *fscanf* is the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure. However, if an input failure occurs before any conversion, `EOF` is returned.

SEE ALSO

fopen(2), *fgetc*(2)

BUGS

Does not know about UTF.

NAME

gamma − log gamma function

SYNOPSIS

```
double gamma(double x)

int    signgam;
```

DESCRIPTION

Gamma returns ln $|\Gamma(x)|$. The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

EXAMPLES

Computation of the gamma function:

```
errno = 0;
y = gamma(x);
if(errno || y > 88)
        error();
y = signgam*exp(y);
```

SEE ALSO

intro(2)

NAME

getenv, putenv − access environment variables

SYNOPSIS

```
char* getenv(char *name)
int   putenv(char *name, char *val)
```

DESCRIPTION

Getenv reads the contents of `/env/`*name* (see *env*(3)) into memory allocated with *malloc*(2), 0-terminates it, and returns a pointer to that area. If no file exists, 0 is returned.

Putenv creates the file `/env/`*name* and writes the string *val* to it. The terminating `0` is not written. If the file value cannot be written, −1 is returned.

SEE ALSO

env(3)

DIAGNOSTICS

Sets *errstr.*

NAME

getfcr, setfcr, getfsr, setfsr − control floating point

SYNOPSIS

```
ulong getfcr(void)

void setfcr(ulong fcr)

ulong getfsr(void)

void setfsr(ulong fsr)
```

DESCRIPTION

These routines provide a fairly portable interface to control the rounding and exception characteristics of IEEE 754 floating point units. In effect, they define a pair of pseudo-registers, the floating point control register, `fcr`, which affects rounding, precision, and exceptions, and the floating point status register, `fsr`, which holds the accrued exception bits. Each register has a *get* routine to retrieve its value, a *set* routine to modify it, and macros that identify its contents.

The `fcr` contains bits that, when set, enable exceptions: `FPINEX` (enable inexact exceptions), `FPOVFL` (enable overflow exceptions), `FPUNFL` (enable underflow exceptions), and `FPZDIV` (enable zero divide exceptions). Rounding is controlled by installing in `fcr`, under mask `FPRMASK`, one of the values `FPRNR` (round to nearest), `FPRZ` (round towards zero), `FPRPINF` (round towards positive infinity), and `FPRNINF` (round towards negative infinity). Precision is controlled by installing in `fcr`, under mask `FPPMASK`, one of the values `FPPEXT` (extended precision), `FPPSGL` (single precision), and `FPPDBL` (double precision).

The `fsr` holds the accrued exception bits `FPAINEX`, `FPAOVFL`, `FPAUNFL`, and `FPAZDIV`, corresponding to the `fsr` bits without the `A` in the name.

Not all machines support all modes. If the corresponding mask is zero, the machine does not support the rounding or precision modes. On some machines it is not possible to clear selective accrued exception bits; a *setfsr* clears them all. The exception bits defined here work on all architectures.

The default state of the floating point unit is fixed for a given architecture but is undefined across Plan 9: the default is to provide what the hardware does most efficiently. Use these routines if you need guaranteed behavior. Also, gradual underflow is not available on some machines.

EXAMPLE

To enable overflow traps and make sure registers are rounded to double precision (for example on the MC68020, where the internal registers are 80 bits long):

```
ulong fcr;
fcr = getfcr();
fcr |= FPOVFL;
fcr &= ~FPPMASK
fcr |= FPPDBL;
setfcr(fcr);
```

NAME

getfields, getmfields, setfields − break a string into fields

SYNOPSIS

```
int   getfields(char *str, char **ptrs, int nptrs)

int   getmfields(char *str, char **ptrs, int nptrs)

char* setfields(char *fielddelim)
```

DESCRIPTION

Getfields breaks the null-terminated string *str* into at most *nptrs* null-terminated fields and places pointers to the start of these fields in the array *ptrs*. It returns the number of fields and terminates the list of pointers with a zero pointer. It overwrites some of the bytes in *str*. If there are *nptr* or more fields, the list will not end with zero and the last 'field' will extend to the end of the input string and may contain delimiters.

A field is defined as a maximal sequence of characters not in a set of field delimiters. Adjacent fields are separated by exactly one delimiter. No field follows a delimiter at the end of string. Thus a string of just two delimiter characters contains two empty fields, and a nonempty string with no delimiters contains one field.

Getmfields is the same as *getfields* except that fields are separated by maximal strings of field delimiters rather than just one.

Setfields makes the field delimiters (space and tab by default) be the characters of the string *fielddelim* and returns a pointer to a string of the previous delimiters.

SEE ALSO

strtok in *strcat*(2)

NAME

getpid, getppid − get process ids

SYNOPSIS

```
int getpid(void)

int getppid(void)
```

DESCRIPTION

Getpid reads `/dev/pid` (see *cons*(3)) and converts it to get the process id of the current process, a number guaranteed to be unique among all running processes on the machine executing *getpid.*

Getppid reads `/dev/ppid` (see *cons*(3)) and converts it to get the id of the parent of the current process.

SEE ALSO

intro(2), *cons*(3), *proc*(3)

DIAGNOSTICS

Returns 0 and sets *errstr* if unsuccessful.

NAME

getuser − get user name

SYNOPSIS

`char* getuser(void)`

DESCRIPTION

Getuser returns a pointer to static data which contains the name of the user who owns the current process. *Getuser* reads `/dev/user` to find the name.

SEE ALSO

intro(2), *cons*(3)

NAME

getwd − get current directory

SYNOPSIS

```
char* getwd(char *buf, int size)
```

DESCRIPTION

Getwd will fill *buf* with a null-terminated string representing the current directory and return *buf*.

Getwd will place no more than *size* bytes in the buffer provided.

SEE ALSO

pwd(1)

DIAGNOSTICS

On error, zero is returned and *buf* is filled with a diagnostic message. *Errstr*(2) may be consulted for more information.

NAME

Point, Rectangle, Bitmap, Cursor, binit, bclose, berror, bscreenrect, bneed, bflush, bwrite, bexit, clipr, cursorswitch, cursorset, rdfontfile, ffree, charwidth, Pconv, Rconv − graphics

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>

void      binit(void (*errfun)(char *), char *font, char *label)

void      bclose(void)

void      bexit(void)

void      berror(char *msg)

Rectangle bscreenrect(Rectangle *clipr)

uchar*    bneed(int n)

void      bflush(void)

int       bwrite(void)

int       clipr(Bitmap *b, Rectangle cr)

void      cursorswitch(Cursor *curs)

void      cursorset(Point p)

Font*     rdfontfile(char *name, int ldepth)

void      ffree(Font *f)

int       charwidth(Font *f, Rune r)

int       Pconv(void *arg, int f1, int f2, int f3, int chr)

int       Rconv(void *arg, int f1, int f2, int f3, int chr)

extern Bitmap    screen

extern Font      *font
```

DESCRIPTION

A `Point` is a location in a bitmap (see below), such as the screen, and is defined as:

```
typedef
struct Point {
     int x;
     int y;
} Point;
```

The coordinate system has *x* increasing to the right and *y* increasing down.

A `Rectangle` is a rectangular area in a bitmap.

```
typedef
struct Rectangle {
     Point min;      /* upper left */
     Point max;      /* lower right */
} Rectangle;
```

By definition, `min.x <= max.x` and `min.y <= max.y`. By convention, the right (maximum *x*) and bottom (maximum *y*) edges are excluded from the represented rectangle, so abutting rectangles have no points in common. Thus, `max` contains the coordinates of the first point beyond the rectangle.

A `Bitmap` holds a rectangular image.

```
typedef
struct Bitmap {
     Rectangle r;       /* rectangle in data area, local coords */
     Rectangle clipr;   /* clipping region */
     int       ldepth;  /* log base 2 of number of bits per pixel */
     int       id;      /* id as known in /dev/bitblt */
     Bitmap*   cache;   /* zero; distinguishes bitmap from layer */
} Bitmap;
```

`R.min` is the location in the bitmap of the upper-leftmost point in the image. There are 2^{ldepth} contiguous bits for each pixel of the image; the bits form a binary number giving the pixel value. `Clipr` is the clipping rectangle; typically it is the same as `r` except in a window, where it is inset by the width of the border. Graphical operations on the `Bitmap` will be confined to the clipping rectangle. The subroutine *Clipr* sets the clipping rectangle of `b` to the intersection of `cr` and `b->r`. If *cr* does not intersect `b->r` it does nothing. *Clipr* returns 1 if the clipping region was set, 0 if it was not.

A `Font` is a set of character images, indexed by runes (see *utf*(6)). The images are organized into `Subfonts`, each containing the images for a small, contiguous set of runes. `Font` and `Subfont` structures contain two related fields: `ascent`, the distance from the top of the highest character (actually the top of the bitmap holding all the characters) to the baseline, and `height`, the distance from the top of the highest character to the bottom of the lowest character (and hence, the interline spacing). The width of any particular character `r` in a font is returned by *charwidth.* The width is defined as the amount to add to the horizontal position after drawing the character. `Charwidth` calls the graphics error function if `r` is zero (NUL) because `string` (see *bitblt*(2)) cannot draw a NUL. The other fields are used internally by the text-drawing functions. See *cachechars*(2) for a detailed description.

Rdfontfile reads the font description in file *name* and returns a pointer that can by used by *string* (see *bitblt*(2)) to draw characters from the font. The *ldepth* argument specifies how characters will be cached; it should usually be the ldepth of the bitmap that will most often be the target of *string*. *Ffree* frees a font. The convention for naming font files is:

`/lib/font/bit/`*name*`/`*range*`.`*size*`.font`

where *size* is approximately the height in pixels of the lower case letters (without ascenders or descenders). *Range* gives some indication of which characters will be available: for example `ascii`, `latin1`, `euro`, or `unicode`. `Euro` includes most European languages, punctuation marks, the International Phonetic Alphabet, etc., but no Oriental languages. `Unicode` includes every character for which images exist on the system.

A *Cursor* is defined:

```
typedef struct
Cursor {
     Point offset;
     uchar clr[2*16];
     uchar set[2*16];
} Cursor;
```

The arrays are arranged in rows, two bytes per row, left to right in big-endian order to give 16 rows of 16 bits each. A cursor is displayed on the screen by adding `offset` to the current mouse position, using `clr` as a mask to zero the pixels where `clr` is 1, and then setting pixels to ones where `set` is one.

The function *binit* must be called before using any graphics operations. The *errfun* argument is a function to be called with an error message argument when the graphics functions detect a fatal error; such an error function must not return. A zero for the *errfun* specifies the default *berror*, which prints the message and exits. If *label* is non-null, it will be written to `/dev/label`, so that it can be used to identify the window when hidden (see *8½*(1)). `Binit` sets up the global *screen* to be a bitmap describing the area of the screen that the program can use. This will be either the whole screen, or some portion of it if the program is running under a window system such as *8½*(1). `Binit` also establishes a font by reading the named *font* file. If `font` is null, *binit* reads the file named in the environment variable `$font`; if `$font` is not set, it imports the default (usually minimal) font from the operating system. The global *font* will be set to point to the resulting `Font` structure. Another effect of *binit* is that it installs *print*(2) formats *Pconv* and *Rconv* as `%P` and `%R` for printing `Points` and `Rectangles`.

Bclose closes the file descriptor connecting the application to the graphics server, typically for use by a child process that needs to disconnect from the graphics server. It does not automatically flush pending output (see *bflush*, below). *Bclose* is not needed by most programs. *Bexit* completes any pending graphics. It is called automatically by *exits*(2).

The *screen*`.r` field is not maintained across 'reshape' events; use *bscreenrect* to discover the current size (see *event*(2)); a non-null `cr` will be filled in with the screen's clip rectangle.

The mouse cursor is always displayed. The initial cursor is an arrow. *Cursorswitch* causes the argument cursor to be displayed instead. A zero argument causes a switch back to the arrow cursor. *Cursorset* moves the mouse cursor to position *p,* provided (if in a window) that the requesting program is executing in the current window and the mouse is within the window boundaries; otherwise *cursorset* is a no-op.

The graphics functions described in *bitblt*(2), *balloc*(2), *cachechars*(2), and *subfalloc*(2) are implemented by writing commands to `/dev/bitblt` (see *bit*(3)); the writes are buffered, so the functions may not take effect immediately. *Bflush* flushes the buffer, doing all pending graphics operations. *Binit* arranges that *bflush* will be called on exit, and the following functions all cause a flush: *balloc, bfree, bscreenrect, cursorset, cursorswitch, ecankbd, ecanmouse, ekbd, emouse, event, rdfontfile, subfalloc, ffree, rdbitmap,* and *wrbitmap.*

The rare program that needs to implement the `/dev/bitblt` protocol directly can use *bneed* and *bwrite. Bneed* returns a pointer to a place in the write buffer, allocating space for *n* bytes. The buffer will be flushed first if *n* is zero, or the buffer is too full. After filling in bytes allocated with *bneed*, *bwrite* can be used to write everything in the buffer and reset the buffer pointer. Unlike *bflush, bwrite* does not call the registered error function and so can be used when an error is possible and the error function is inappropriate.

FILES

`/lib/font/bit` directory of bitmap fonts

SEE ALSO

add(2), *balloc*(2), *cachechars*(2), *subfalloc*(2), *bitblt*(2), *event*(2), *frame*(2), *print*(2), *bit*(3), *layer*(2), *bitmap*(6), *font*(6)

DIAGNOSTICS

An error function may call *errstr*(2) for further diagnostics.

NAME

hypot − Euclidean distance

SYNOPSIS

```
double hypot(double x, double y)
```

DESCRIPTION

Hypot returns

```
sqrt(x*x + y*y)
```

taking precautions against unwarranted overflows.

NAME

eipconv, parseip, parseether, myipaddr, myetheraddr, maskip, etherip, equivip − Internet protocol

SYNOPSIS

```
#include <ip.h>

int   eipconv(void *o, int f1, int f2, int f3, int chr)

int   parseip(uchar *ipaddr, char *str)

int   parseether(uchar *eaddr, char *str)

int   myipaddr(uchar *ipaddr, char *net)

int   myetheraddr(uchar *eaddr, char *net)

void  maskip(uchar *from, uchar *mask, uchar *to)

int   equivip(uchar *ipaddr1, uchar *ipaddr2)
```

DESCRIPTION

These routines are used by Internet Protocol (IP) programs to manipulate IP and Ethernet addresses. IP addresses are stored as a string of 4 `unsigned chars`, Ethernet addresses as 6 `unsigned chars`. The string representation of IP addresses is (up to) 4 decimal integers from 0 to 255 separated by periods. The string representation of Ethernet addresses is exactly 12 hexadecimal digits.

Eipconv is a *print*(2) formatter for Ethernet (verb `I`) and Internet protocol (verb `E`) addresses.

Parseip converts a string pointed to by *str* to a 4-byte IP address starting at *ipaddr. Myipaddr* reads the IP address string from file /*net*/`1/local` and parses it into *ipaddr*. Both routines return a negative number on errors.

Parseether converts a string pointed to by *str* to a 6 byte Ethernet address starting at *eaddr. Myetheraddr* reads the Ethernet address string from file *net*`/1/stats` and parses it into *eaddr*. Both routines return a negative number on errors.

Maskip places the bit-wise AND of the IP addresses pointed to by its first two arguments into the buffer pointed to by the third.

Equivip returns non-zero if the IP addresses pointed to by its two arguments are equal.

SEE ALSO

print(2)

NAME

lalloc, lfree, ltofront, ltoback, lcstring − graphics layers

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>
#include <layer.h>

Layer*      lalloc(Cover *c, Rectangle r)

void   lfree(Layer *l)

void   ltofront(Layer *l)

void   ltoback(Layer *l)

void   lcstring(Bitmap *b, int height, uchar *widths, uchar *msg, int n)
```

DESCRIPTION

The layer library extends the functionality of the bitmap graphics library (see *graphics*(2)) to overlapping independent rectangular windows, or *layers,* on a single bitmap, typically the screen. The entry points `bitblt`, `point`, `segment`, `string`, `subfontstring`, and `texture` are overloaded in the layer library to apply these routines equally to bitmaps and layers. Other than *lcstring,* which is rarely needed, there are no special entry points for drawing on layers.

The data structures associated with the main type, `Layer`, are defined in `<layer.h>`:

```
typedef struct Layer Layer;
typedef struct Cover Cover;
typedef enum Lvis {
     Visible,
     Obscured,
     Invisible,
}Lvis;

struct Layer {
     Bitmap;            /* Bitmap.cache!=0 ==> layer */
     Layer  *next;      /* next layer from front to back */
     Cover  *cover;     /* layer etc. from which this is derived */
     int    user;       /* a place for the user to stick stuff */
     Lvis   vis;        /* visibility state */
};

struct Cover {
     Layer  *layer;     /* layer on which these are painted */
     Layer  *front;     /* first sublayer */
     Bitmap *ground;    /* background texture */
};
```

`Layers` and `Bitmaps` are distinguished by the `cache` element of their structures: `cache` is non-zero in a `Layer`. The layer library's versions of the graphics routines listed above use `cache` to decide how to implement their operations. These functions operate on type `Bitmap*` but because `Bitmap` is included in `Layer`, the C compiler will permit passing a `Layer` to these routines. The routines promote the type to `Layer*` if they see `cache` is non-zero. (Note that these actions apply only in the layer library; although `cache` is defined in `Bitmaps`, the standard graphics library does not support layers.)

Lalloc allocates a new `Layer` to occupy `Rectangle` *r* in a `Bitmap`. The argument `Cover` *c* connects the set of `Layers` to a *covering* `Bitmap`. Before the first call to *lalloc, c* should be allocated and initialized so *c*`->cover` is the `Bitmap` on which the `Layers` will be drawn, *c*`->front` is zero, *c*`->ground` is a background texture to fill the interstices between `Layers`, and *c*`->cover` is textured with *c*`->ground`. It is legal for *c*`->cover` itself to be a `Layer` for recursive layering. The rectangle *r* may have arbitrary overlap, including none, with the *c*`->cover->r`. After calling *lalloc,* the new `Layer` is fully visible (as far as geometry permits) on the covering `Bitmap` and is cleared to all zeros.

Lfree frees the `Layer` *l* and restores the contents of its covering `Bitmap`.

Ltofront makes *l* fully visible within its covering `Bitmap`. *Ltoback* pushes *l* behind any other `Layers` on the same covering `Bitmap`. Neither function changes the x-y location of the `Layer`.

Lcstring is peculiar to programs, such as *8½*(1), that multiplex client access to the display. It acts as a feed-through for the `'s'` message generated by `string` (see *bit*(3)). *B* is the bitmap (or layer) and *height* is the height of the font in which the string is to be drawn. *Widths* is an array of character widths, indexed by font cache position. *Msg* is a pointer to the `string` message; it contains the header and *n* cache indices.

SEE ALSO

graphics(2), *bitblt*(2), *cachechars*(2), *bit*(3)

NAME

crackhdr, newmap, setmap, unusemap, freemap, loadmap, mget, mput, beswab, beswal, leswab, leswal − executable file interpretation

SYNOPSIS

```
#include <bio.h>
#include <mach.h>

int  crackhdr(int fd, Fhdr *fp)

Map  *newmap(Map *map, int fd)

int  setmap(Map *map, int seg, ulong base, ulong end, ulong foffset)

void unusemap(Map *map, int seg)

Map  *loadmap(Map *map, int fd, Fhdr *fp)

int  mget(Map *map, int seg, ulong addr, char *buf, int size)

int mput(Map *map, int seg, ulong addr, char *buf, int size)

ushort beswab(ushort s)

long beswal(long l)

ushort leswab(ushort s)

long leswal(long l)
```

DESCRIPTION

These functions provide machine-independent processing of an executable file or executing process image. The latter is accessible by opening the device `/proc/`*pid*`/text` as described in *proc*(3). The functions are stored in library `libmach.a`; the library is automatically searched by the loader when header file *mach.h* is included in a source file. *Symbol*(2) and *object*(2) describe additional library functions for processing symbol tables and object files.

Crackhdr loads data structure *fp* with a machine-independent description of the header of the executable file or image associated with the open file descriptor *fd*. It also sets global variable *mach* pointing to the `Mach` data structure containing the machine-dependent parameters of the target architecture.

A *Map* is a data structure used to transform an address in the logical address space of an executable to an offset in a file or executing image. Each map comprises up to four logical `segments`, named `SEGDATA`, `SEGTEXT`, `SEGUBLK`, and `SEGREGS`, that map the data, text, `u-block`, and register segments, respectively. The latter two segments are only applicable to executing images. A portion of the physical address space may be mapped by multiple segments. A segment defines the low and high addresses of the logical address space and the physical offset into the file or executing image to the beginning of the address space.

Newmap creates a new map or recycles one currently in use. If *map* is zero, a new map is dynamically allocated, otherwise it is assumed to point to an existing map. The map is marked empty and attached to the open file descriptor *fd*. The address of the map is returned.

Setmap loads segment *seg* of *map* with the segment mapping parameters. *Base* and *end* contain the lowest and highest virtual addresses mapped by the segment. *Foffset* contains the offset in the executable to the start of the segment.

Unusemap marks segment *seg* in map *map* unused. Other segments in the map remain unaffected.

Loadmap uses the values in a `Fhdr` data structure (usually filled by *crackhdr*) to initialize the map for an executable file or executing image. If *map* is zero, a new map is dynamically allocated; otherwise, *map* is initialized with the appropriate values. This function returns the address of the map if successful, zero on failure.

Mget reads *size* bytes into *buf* from the file associated with *map*. The data is read from logical address *addr* in segment *seg*. *Fput* is similar except it writes to the executable file or executing image associated with *map*. Both functions return −1 if they are unable to calculate a physical address, 0 if the read or write operation fails, and 1 on success. The segment is one of `SEGTEXT`, `SEGDATA`, `SEGUBLK`, or `SEGREGS`, or the special segment, `SEGANY`. If `SEGANY` is specified, the address translation is performed using the text, data, and `u-block` maps, in that order. Accesses to `SEGDATA` first attempt a translation using the data map then the `u-block` map. The read or write operation takes place at the address produced by the first valid translation.

Beswab and *beswal* convert a big-endian `ushort` and `long` respectively, to the target processor's native representation. *Leswab* and *leswal* perform the same conversion for a little-endian `ushort` and `long` respectively.

Unless otherwise specified, all functions return 1 on success, or 0 on error.

SEE ALSO

symbol(2), *object*(2), *proc*(3), *a.out*(6)

NAME

malloc, free, realloc, calloc, mstats − memory allocator

SYNOPSIS

```
void* malloc(long size)

void  free(void *ptr)

void* realloc(void *ptr, long size)

void* calloc(long nelem, long elsize)
```

DESCRIPTION

Malloc and *free* provide a simple memory allocation package. *Malloc* returns a pointer to a new block of at least *size* bytes. The block is suitably aligned for storage of any type of object. No two active pointers from *malloc* will have the same value.

The argument to *free* is a pointer to a block previously allocated by *malloc*; this space is made available for further allocation. It is legal to free a null pointer; the effect is a no-op.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. The call `realloc(0, size)` means the same as `malloc(size)`.

Calloc allocates space for an array of *nelem* elements of size *elsize.* The space is initialized to zeros. *Free* frees such a block.

SEE ALSO

brk(2)

DIAGNOSTICS

Malloc, realloc and *calloc* return 0 if there is no available memory. *Errstr* is likely to be set.

BUGS

The different specification of *calloc* is bizarre.

User errors can corrupt the storage arena. The most common gaffes are (1) freeing an already freed block, (2) storing beyond the bounds of an allocated block, and (3) freeing data that was not obtained from the allocator. When *malloc* and *free* detect such corruption, they abort.

NAME

memccpy, memchr, memcmp, memcpy, memmove, memset − memory operations

SYNOPSIS

```
void* memccpy(void *s1, void *s2, int c, long n)

void* memchr(void *s, int c, long n)

int   memcmp(void *s1, void *s2, long n)

void* memcpy(void *s1, void *s2, long n)

void* memmove(void *s1, void *s2, long n)

void* memset(void *s, int c, long n)
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of bytes bounded by a count, not terminated by a zero byte). They do not check for the overflow of any receiving memory area.

Memccpy copies bytes from memory area *s2* into *s1*, stopping after the first occurrence of byte *c* has been copied, or after *n* bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of *c* in *s1*, or zero if *c* was not found in the first *n* bytes of *s2*.

Memchr returns a pointer to the first occurrence of byte *c* in the first *n* bytes of memory area *s,* or zero if *c* does not occur.

Memcmp compares its arguments, looking at the first *n* bytes only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*. The comparison is bytewise unsigned.

Memcpy copies *n* bytes from memory area *s2* to *s1.* It returns *s1.*

Memmove works like *memcpy*, except that it is guaranteed to work if *s1* and *s2* overlap.

Memset sets the first *n* bytes in memory area *s* to the value of byte *c*. It returns *s*.

SEE ALSO

strcat(2)

BUGS

ANSI C does not require *memcpy* to handle overlapping source and data; on Plan 9, it does, so *memmove* and *memcpy* behave identically.

If *memcpy* and *memmove* are handed a negative count, they abort.

NAME

mktemp − make a unique file name

SYNOPSIS

`char* mktemp(char *template)`

DESCRIPTION

Mktemp replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with eleven trailing `X`s. The `X`s are replaced by a letter followed by the current process id. Letters from `a` to `z` are tried until a name that can be accessed (see *access*(2)) is generated. If no such name can be generated, *mktemp* returns `"/"`.

SEE ALSO

getpid(2), *access*(2)

NAME

NaN, Inf, isNaN, isInf − not-a-number and infinity functions

SYNOPSIS

```
double NaN(void)

long   Inf(int)

int    isNaN(double)

int    isInf(double, int)
```

DESCRIPTION

The IEEE floating point standard defines values called ‘not-a-number’ and positive and negative ‘infinity’. These values can be produced by such things as overflow and division by zero. Also, the library functions sometimes return them when the arguments are not in the domain, or the result is out of range.

NaN returns a double that is not-a-number. *IsNaN* returns true if its argument is not-a-number.

Inf(*i*) returns positive infinity if *i* is greater than or equal to zero, else negative infinity. *IsInf* returns true if its first argument is infinity with the same sign as the second argument.

NAME

ndbopen, ndbclose, ndbreopen, ndbsearch, ndbsnext, ndbgetval, ndbfree, ipattr, ipinfo, ndbhash, ndbseek, ndbparse − network database

SYNOPSIS

```
#include <bio.h>
#include <ndb.h>

Ndb*        ndbopen(char *file);

int         ndbreopen(Ndb *db);

void        ndbclose(Ndb *db);

Ndbtuple*   ndbsearch(Ndb *db, Ndbs *s, char *attr, char *val);

Ndbtuple*   ndbsnext(Ndbs *s, char *attr, char *val);

Ndbtuple*   ndbgetval(Ndb *db, Ndbs *s, char *attr, char *val,
            char *rattr, char *buf);

void        ndbfree(Ndbtuple *db);

char*       ipattr(char *name);

int         ipinfo(Ndb *db, char *ether, char *ip, char *name,
            Ipinfo *iip);

ulong       ndbhash(char *val, int hlen);

long        ndbseek(Ndb *db, long off, int whence);

Ndbtuple*   ndbparse(Ndb *db);
```

DESCRIPTION

These routines are used by network administrative programs to search the network database. They operate on the database files described in *ndb*(6).

Ndbopen opens the database *file* and calls *malloc*(2) to allocate a buffer for it. If *file* is zero, all network database files are opened.

Ndbreopen checks if the database files associated with *db* have changed and if so throws out any cached information and reopens the files.

Ndbclose closes any database files associated with *db* and frees all storage associated with them.

Ndbsearch and *ndbsnext* search a database for an entry containing the attribute/value pair, *attr*=*val*. *Ndbsearch* is used to find the first match and *ndbsnext* is used to find each successive match. On a successful search both return a linked list of *Ndbtuple* structures acquired by *malloc*(2) that represent the attribute/value pairs in the entry. On failure they return zero.

```
typedef struct Ndbtuple Ndbtuple;
struct Ndbtuple {
        char       attr[Ndbalen];
        char       val[Ndbvlen];
        Ndbtuple   *entry;
        Ndbtuple   *line;
};
```

The *entry* pointers chain together all pairs in the entry in a null terminated list. The *line* pointers chain together all pairs on the same line in a circular list. Thus, a program can implement 2 levels of binding for pairs in an entry. In general, pairs on the same line are bound tighter than pairs on different lines.

The structure *Ndbs* is used to link successive searches.

```
typedef struct Ndbs Ndbs;
struct Ndbs {
```

```
                Ndb       *db;    /* data base file being searched */
                ...
                Ndbtuple *t;     /* last attribute value pair found */
        };
```

The *t* field points to the pair within the entry matched by the *ndbsearch* or *ndbsnext*.

Ndbgetval searches the database for an entry containing not only an attribute/value pair, *attr*=*val*, but also a pair with the attribute *rattr*. If successful, it copies the value associated with *rattr* into *buf. Buf* must point to an area at least *Ndbvlen* long.

Ndbfree frees a list of tuples returned by one of the other routines.

Ipattr takes the name of an IP system and returns the attribute it corresponds to:

`dom`	domain name
`ip`	Internet number
`sys`	system name

Ipinfo searches the database for Internet Protocol information about a system and returns it in the structure addressed by *iip.* The arguments *ether* (textual Ethernet address), *ip* (textual IP address), and *name* identify the system. At least one must be non-zero. *Ipinfo* returns 0 if successful, −1 otherwise. Both *bootp*(8) and *ipconfig*(8) use *ipinfo* to search the database.

The last three calls are used by programs that create the hash tables and database files. *Ndbhash* computes a hash offset into a table of length *hlen* for the string *val*. *Ndbseek* causes a subsequent read, write, or *ndbparse* of the database file to start at the position specified by the last two arguments. These arguments have the same meaning as the last two arguments of *seek*(2). *Ndbseek* returns a negative number on error. *Ndbparse* reads and parses the next entry from the database file. Multiple *ndbparse*'s without intervening *ndbseek*'s parse sequential entries in the database file. A zero is returned at end of file.

SEE ALSO

ndb(6) *ndb*(8)

NAME

notify, noted, atnotify − handle asynchronous process notification

SYNOPSIS

```
int notify(void (*f)(void*, char*))

int noted(int v)

int atnotify(int (*f)(void*, char*), int in)
```

DESCRIPTION

When a process raises an exceptional condition such as dividing by zero or writing on a closed pipe, a *note* is posted to communicate the exception. A note may also be posted by a *write*(see *read*(2)) to the process's `/proc/`*n*`/note` file or to the `/proc/`*m*`/notepg` file of a process in the same process group (see *proc*(3)). When the note is received the behavior of the process depends on the origin of the note. If the note was posted by an external process, the process receiving the note exits; if generated by the system the note string, preceded by the name and id of the process and the string `"suicide: "`, is printed on the process's standard error file and the process is suspended in the `Broken` state for debugging.

These default actions may be overridden. The *notify* function registers a *notification handler* to be called within the process when a note is received. The argument to *notify* replaces the previous handler, if any. An argument of zero cancels a previous handler, restoring the default action. A *fork*(2) system call leaves the handler registered in both the parent and the child; *exec*(2) restores the default behavior.

After a note is posted, the handler is called with two arguments: the first is a pointer to a `Ureg` structure (defined in `/$objtype/include/ureg.h`) giving the current values of registers; the second is a pointer to the note itself, a null-terminated string with no more than `ERRLEN` characters in it including the terminal NUL. The `Ureg` argument is usually not needed; it is provided to help recover from traps such as floating point exceptions. Its use and layout are machine- and system-specific.

A notification handler must finish by calling *noted*; if the handler returns the behavior is undefined and probably erroneous. The argument to *noted* defines the action to take: `NDFLT` instructs the system to perform the default action as if the handler had never been registered; `NCONT` instructs the system to resume the process at the point it was notified. In neither case does *noted* return to the handler. If the note interrupted an incomplete system call, that call returns an error (with error string `interrupted`) after the process resumes. A notification handler can also jump out to an environment set up with *setjmp* using the *notejmp* function (see *setjmp*(2)).

Regardless of the origin of the note or the presence of a handler, if the process is being debugged (see *proc*(3)) the arrival of a note puts the process in the `Stopped` state and awakens the debugger.

Rather than using the system calls *notify* and *noted*, most programs should use *atnotify* to register notification handlers. The parameter *in* is non-zero to register the function *f*, and zero to cancel registration. A handler must return a non-zero number if the note was recognized (and resolved); otherwise it must return zero. When the system posts a note to the process, each handler registered with *atnotify* is called with arguments as described above until one of the handlers returns non-zero. Then *noted* is called with argument `NCONT.` If no registered function returns non-zero, *atnotify* calls *noted* with argument `NDFLT`.

The set of notes a process may receive is system-dependent, but there is a common set that includes:

Note	*Meaning*
`interrupt`	user interrupt (DEL key)
`hangup`	I/O connection closed
`alarm`	alarm expired
`sys: breakpoint`	breakpoint instruction
`sys: bad address`	system call address argument out of range
`sys: odd address`	system call address argument unaligned
`sys: bad sys call`	system call number out of range
`sys: odd stack`	system call user stack unaligned
`sys: write on closed pipe`	write on closed pipe

`sys: fp:` *fptrap*	floating point exception
`sys: trap:` *trap*	other exception (see below)

The notes prefixed `sys:` are generated by the operating system. They are suffixed by the user program counter in format `pc=0x1234`. If the note is due to a floating point exception, just before the `pc` is the address of the offending instruction in format `fppc=0x1234`. Notes are limited to `ERRLEN` bytes; if they would be longer they are truncated but the `pc` is always reported correctly.

The types and syntax of the *trap* and *fptrap* portions of the notes are machine-dependent.

SEE ALSO

intro(2), *notejmp* in *setjmp*(2)

BUGS

Since *exec*(2) discards the notification handler, there is a window of vulnerability to notes in a new process.

NAME

objtype, readobj, objsym, objbase, objreset, isar, nextar, readar − object file interpretation functions

SYNOPSIS

```
#include <bio.h>
#include <mach.h>

int  objtype(Biobuf *bp)

int  readobj(Biobuf *bp, int objtype)

Sym  *objsym(int index)

Sym  *objbase(long *nsyms)

void  objreset()

int isar(Biobuf *bp)

int nextar(Biobuf *bp, int offset, char *buf)

int readar(Biobuf *bp, int objtype, int end)
```

DESCRIPTION

These functions provide machine-independent access to object files stored in a directory or contained in an archive. They are contained in library *libmach.a*; the library is automatically searched by the loader when header file *mach.h* is included in a source file. *Mach*(2) and *symbol*(2) describe additional library functions for interpreting executable files and executing images.

Object files contain no formal symbol table; instead, references to symbols must be extracted from the encoded object representation and resolved. The resulting symbol information is added to a dummy symbol table where it may be processed by an application. The organization of the internal symbol table is identical to that produced by the loader and described in *symbol*(2)*and a.out*(6); a vector of `Sym` data structures defining the name, type and relative offset of each symbol.

Objtype reads the header at the current position of the file associated with *bp* (see *Bio*(2)) and returns a code indicating the target architecture of the file or -1 if the type cannot be discerned. The file may be a stand-alone object file or a member of an archive. The position of the file is rewound to its current position following the decoding of the header.

Readobj constructs a symbol table for the object file associated with *bp*. The second argument contains the type code produced by function *objtype*. The file must be positioned at the start of the object file. Multiple invocations of *readobj* append the symbol definitions for each object file to the existing symbol table. *Objreset* can be used to clear a symbol table.

Objsym returns the address of the *i*th `Sym` structure in the symbol table or zero if *index* is out of range.

Objbase returns the address of the first *Sym* structure in the symbol table. The number of entries in the symbol table is returned in *nsyms*. *Readobj*or*readar* must be invoked prior to *symbase*and*symobj* to build the symbol table.

Objreset clears the internal symbol table built by *readobj* or *readar*.

Isar reads the header at the current point in the file associated with *bp* and returns 1 if it is an archive or zero otherwise. The file is left positioned at the end of the archive header and at the beginning of the first member of the archive.

Nextar extracts information describing the archive member stored at *offset* in the file associated with *bp*. If the header describing the member can be extracted and decoded, the size of the member is returned. Adding this value to *offset* yields the offset of the beginning of the next member in the archive. On return the input file is positioned at the end of the member header immediately before the first byte of the archive and the name of the member is stored in *buf*, a buffer of `NNAME` characters. If there are no more members, *nextar* returns zero; a negative return indicates a missing or malformed header.

Readar constructs the symbol table of the object file stored at the current position in the archive associated with *bp*. This function operates exactly as *readobj*; the only difference is the extra argument, *end*, specifying the offset to the beginning of the next member in the archive. Following execution the file is positioned at the beginning of the member header of the next member.

SEE ALSO

mach(2), *symbol*(2), *bio*(2), *a.out*(6)

NAME

open, create, close − open a file for reading or writing, create file

SYNOPSIS

```
int open(char *file, int omode)

int create(char *file, int omode, ulong perm)

int close(int fd)
```

DESCRIPTION

Open opens the *file* and returns an associated file descriptor. *Omode* is one of `OREAD`, `OWRITE`, `ORDWR`, or `OEXEC`, asking for permission to read, write, read and write, or execute, respectively. In addition, there are three values that can be ORed with the omode: `OTRUNC` says to truncate the file to zero length before opening it; `OCEXEC` says to close the file when an *exec*(2) or *execl* system call is made; and `ORCLOSE` says to remove the file when it is closed (by everyone who has it open). The *omode* values are defined in `<libc.h>`. *Open* fails if the file does not exist or the user does not have permission to open it for the requested purpose (see *stat*(2) for a description of permissions). The user must have write permission on the *file* if the `OTRUNC` bit is set. For the *open* system call (unlike the implicit *open* in *exec*(2)), `OEXEC` is actually identical to `OREAD`.

Create creates a new *file* or prepares to rewrite an existing *file*, opens it according to *omode* (as described for *open*), and returns an associated file descriptor. If the file is new, the owner is set to the userid of the creating process group; the group to that of the containing directory; the permissions to *perm* ANDed with the permissions of the containing directory. If the file already exists, it is truncated to 0 length, and the permissions, owner, and group remain unchanged. The created file is a directory if the `CHDIR` bit is set in *omode*. *Create* fails if the path up to the last element of *file* cannot be evaluated, if the user doesn't have write permission in the final directory, or if the file already exists and does not permit the access defined by *omode.* If the file is new and the directory in which it is created is a union directory (see *intro*(2)) then the constituent directory where the file is created depends on the structure of the union: see *bind*(2).

Close closes the file associated with a file descriptor. Provided the file descriptor is a valid open descriptor, *close* is guaranteed to close it; there will be no error. Files are closed upon termination of a process; *close* allows the file descriptor to be reused.

SEE ALSO

intro(2), *bind*(2), *stat*(2)

DIAGNOSTICS

These functions set *errstr.*

NAME

perror, syslog − system error messages

SYNOPSIS

```
void perror(char *s)

void syslog(int cons, char *logname, char *fmt, ...)
```

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system. First the argument string *s* is printed, then a colon, then the message and a new-line. If *s* is 0, only the error message and new-line are printed.

Syslog logs messages in the file named by *logname* in the directory `/sys/log`; the file must already exist and should be append-only. *Logname* must contain no slashes. The message is a line with up to five fields: the current time; the program name (if *argv0* is set; see *ARG*(2)); the user name; the message specified by the *print*(2) format *fmt* and any following arguments; and a final newline. If *cons* is set or the log file cannot be opened, the message is also printed on the system console. *Syslog* can be used safely in multi-threaded programs.

SEE ALSO

intro(2), *errstr*(2)

NAME

pipe − create an interprocess channel

SYNOPSIS

`int pipe(int fd[2])`

DESCRIPTION

Pipe creates a buffered channel for interprocess I/O communication. Two file descriptors are returned in *fd*. Data written to `fd[1]` is available for reading from `fd[0]` and data written to `fd[0]` is available for reading from `fd[1]`.

After the pipe has been established, cooperating processes created by subsequent *fork*(2) calls may pass data through the pipe with *read* and *write* calls. The bytes placed on a pipe by one *write* are contiguous even if many processes are writing. Write boundaries are preserved: each read terminates when the read buffer is full or after reading the last byte of a write, whichever comes first.

The number of bytes available to a *read*(2) is reported in the `Length` field returned by *fstat* or *dirfstat* on a pipe (see *stat*(2)).

SEE ALSO

intro(2), *read*(2), *pipe*(3)

DIAGNOSTICS

Sets *errstr.*

BUGS

If a read or a write of a pipe is interrupted, some unknown number of bytes may have been transferred.

NAME

postnote − send a note to a process or process group

SYNOPSIS

`int   postnote(int pid, char *note)`

DESCRIPTION

Postnote sends a note to a process or process group. If *pid* is positive, *note* is written to `/proc/`*pid*`/note.` If it is negative, the note is delivered to the process group by writing *note* to `/proc/`−*pid*`/notepg.`

If the write is successful, zero is returned. Otherwise −1 is returned.

SEE ALSO

notify(2), *intro*(2), *proc*(3)

DIAGNOSTICS

Sets *errstr.*

NAME

print, fprint, sprint, snprint, fmtinstall, strconv, Strconv, numbconv, fltconv, doprint, donprint − print formatted output

SYNOPSIS

```
int   print(char *format, ...)

int   fprint(int fd, char *format, ...)

int   sprint(char *s, char *format, ...)

int   snprint(char *s, int len, char *format, ...)

int   fmtinstall(char c, int (*f)(void*, Fconv*))

void  strconv(char *s, Fconv *fp)

void  Strconv(Rune *s, Fconv *fp)

int   numbconv(void *o, Fconv *fp)

int   fltconv(double f, Fconv *fp)

char* doprint(char *s, char *es, char *format, void *argp)

extern int printcol;
```

DESCRIPTION

Print writes text to the standard output. *Fprint* writes to the named output file descriptor; a buffered form is described in *bio*(2). *Sprint* places text followed by the NUL character (`\0`) in consecutive bytes starting at *s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of bytes transmitted (not including the NUL in the case of *sprint*), or a negative value if an output error was encountered. *Snprint* is like *sprint* but is also passed the length of the buffer at *s*.

Each of these functions converts, formats, and prints its trailing arguments under control of a *format* string. The format contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more arguments. The results are undefined if there are arguments of the wrong type or too few arguments for the format. If the format is exhausted while arguments remain, the excess is ignored.

Each conversion specification has the following format:

```
% [flags] verb
```

The verb is a single character and each flag is a single character or a (decimal) numeric string. Up to two numeric strings may be used; the first is called *f1*, the second *f2*. A period can be used to separate them, and if the period is present then *f1* and *f2* are taken to be zero if missing, otherwise they are 'omitted'. Either or both of the numbers may be replaced with the character `*`, meaning that the actual number will be obtained from the argument list as an integer. The flags and numbers are arguments to the *verb* described below.

The numeric verbs `d`, `o`, `x`, and `X` format their arguments in decimal, octal, hexadecimal, and upper case hexadecimal. Each interprets the flags `h`, `l`, `u`, `#`, and – to mean short, long, unsigned, alternate format, and left justified. If neither short nor long is specified, then the argument is an `int`. If unsigned is specified, then the argument is interpreted as a positive number and no sign is output. If two `l` flags are given, then the argument is interpreted as a `vlong` (a 4-byte or sometimes 8-byte integer). If *f2* is not omitted, the number is padded on the left with zeros until at least *f2* digits appear. Then, if alternate format is specified, for `o` conversion, the number is preceded by a `0` if it doesn't already begin with one; for `x` conversion, the number is preceded by `0x`; for `X` conversion, the number is preceded by `0X`. Finally, if *f1* is not omitted, the number is padded on the left (or right, if left justification is specified) with enough blanks to make the field at least *f1* characters long.

The floating point verbs `f`, `e`, `E`, `g`, and `G` take a `double` argument. Each interprets the flags `+`, `-`, and `#` to mean always print a sign, left justified, and alternate format. *F1* is the minimum field width and, if the converted value takes up less than *f1* characters, it is padded on the left (or right, if 'left justified') with spaces. *F2* is the number of digits that are converted after the decimal place for `e`, `E`, and `f` conversions, and *f2* is the maximum number of significant digits for `g` and `G` conversions. The `f` verb produces output of the form `[-]digits[.digits]`. `e` conversion appends an exponent `e[-]digits`, and `E` conversion appends an exponent `E[-]digits`. The `g` verb will output the argument in either `e` or `f` with the goal of producing the smallest output. Also, trailing zeros are omitted from the fraction part of the output, and a trailing decimal point appears only if it is followed by a digit. The `G` verb is similar, but uses `E` format instead of `e`. When alternate format is specified, the result will always contain a decimal point, and for `g` and `G` conversions, trailing zeros are not removed.

The `s` verb copies a string (pointer to `char`) to the output. The number of characters copied (*n*) is the minimum of the size of the string and *f2*. These *n* characters are justified within a field of *f1* characters as described above. The `S` verb is similar, but it interprets its pointer as an array of runes (see *utf*(6)); the runes are converted to UTF before output.

The `c` verb copies a single `char` (promoted to `int`) justified within a field of *f1* characters as described above. The `C` verb is similar, but works on runes.

Fmtinstall is used to install custom verbs and flags. *Fn* should be declared as

```
int   fn(void *o, Fconv *fp)
```

Fn is passed a pointer *o* to whatever argument appears next in the list to *print*. *Fp->chr* is the flag or verb character to cause *fn* to be called; it must have value less than 512. In *fn, fp->f1* and *fp->f2* are the decoded flags in the conversion. A missing *fp->f1* is denoted by the value zero. A missing *fp->f2* is denoted by a negative number. *Fp->f3* is the bitwise OR of all the flags seen since the most recent `%`. The standard flags values are: 1 (+), 2 (−), 4 (#), 8 (`l`), 16 (`h`), 32 (`u`), and 64 (`ll`). If `fp->chr` is a verb, *fn* should return the size of the argument in bytes so *print* can skip over it. If *fp->chr* is a flag, *fn* should return a negative value: the negation of one of the above flag values, or some otherwise unused power of two. All interpretation of *fp->f1*, *fp->f2*, and *fp->f3* is left up to the conversion routine.

Sprint and *snprint* are reentrant; they may be called to help prepare output in custom conversion routines.

Strconv (with a lower-case s) formats a UTF string. *S* is the string, *fp* has the same meaning as above. The *strconv* routine interprets the − flag in fp->f3 as left-justification. *Strconv* (with a capital S) is like *strconv*, but its input is a rune string, which is converted to UTF on output.

Printcol indicates the position of the next output character. Tabs, backspaces and carriage returns are interpreted appropriately.

Numbconv is used to implement the integer verbs; its arguments are like those of the function argument to *fmtinstall*. *Fltconv* is used to implement the floating verbs. Its arguments are like those of the function argument to *fmtinstall*, except that the first argument is the double itself rather than a pointer to it. Both *numbconv* and *fltconv* use *strconv* to put their results into the current print buffer.

One of *strconv, Strconv,* or *numbconv* must be called to produce output; no other routine puts characters in the output buffer.

Doprint formats the arguments starting at *argp* into the buffer starting at *s*, but it writes no characters after the address *es*. It returns a pointer to the NUL terminating the formatted string.

EXAMPLES

This function prints an error message with a variable number of arguments and then quits.

```
void fatal(char *msg, ...)
{
     char buf[1024], *out;

     out = doprint(buf, buf+sizeof(buf), "Fatal error: ");
     out = doprint(out, buf+sizeof(buf), msg, (&msg+1));
     write(2, buf, out-buf);
```

```
          exits("fatal error");
     }
```

This example adds a verb to print complex numbers.

```
     typedef
     struct {
          double    r, i;
     } Complex;

     int
     Xconv(void *v, Fconv *fp)
     {
          char str[50];
          Complex *o;

          o = v;
          sprint(str, "(%g,%g)", o->r, o->i);
          strconv(str, fp);
          return(sizeof(Complex));
     }

     main(...)
     {
          Complex x = (Complex){ 1.5, -2.3 };

          fmtinstall('X', Xconv);
          print("x = %X\n", x);
     }
```

SEE ALSO

fprintf(2), *utf*(6), *errstr*(2)

DIAGNOSTICS

Print and *fprint* set *errstr*.

BUGS

The formatting is close to that specified for ANSI *fprintf*(2); the differences are:

the − flag doesn't work

`u` is a flag here instead of a verb

`X` conversion doesn't use uppercase `A-F` for digits ten to fifteen

there are no `0` or space flags here

there are no `P` or `n` verbs here

Also, and not a bug, *print* and friends generate UTF rather than ASCII.

NAME

qsort − quicker sort

SYNOPSIS

```
void qsort(void *base, long nel, long width,
          int (*compar)(void*, void*))
```

DESCRIPTION

Qsort (quicker sort) sorts an array into nondecreasing order. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of a comparison routine to be called with pointers to elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1)

NAME

rand, lrand, frand, nrand, lnrand, srand − random number generator

SYNOPSIS

```
int    rand(void)

long   lrand(void)

double frand(void)

int    nrand(int val)

long   lnrand(long val)

void   srand(long seed)
```

DESCRIPTION

Rand returns a uniform pseudo-random number *x,* $0 \le x < 2^{15}$.

Lrand returns a uniform `long` *x,* $0 \le x < 2^{31}$.

Frand returns a uniform `double` *x,* $0.0 \le x < 1.0$, This function calls *lrand* twice to generate a number with as many as 62 significant bits of mantissa.

Nrand returns a uniform integer *x,* $0 \le x < val$. *Lnrand* is the same, but returns a `long`.

The algorithm is additive feedback with:

$$x[n] = (x[n-273] + x[n-607]) \bmod 2^{31}.$$

giving a period of $2^{30} \times (2^{607} - 1)$.

The generators are initialized by calling *srand* with whatever you like as argument. To get a different starting value each time,

```
srand(time(0))
```

will work as long as it is not called more often than once per second. Calling

```
srand(1)
```

will initialize the generators to their starting state.

NAME

read, write − read or write file

SYNOPSIS

```
long read(int fd, void *buf, long nbytes)

long write(int fd, void *buf, long nbytes)
```

DESCRIPTION

Read reads *nbytes* bytes of data from the offset in the file associated with *fd* into memory at *buf*. The offset is advanced by the number of bytes read. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to the console, at most one line will be returned. In any event the number of characters read is returned. A return value of 0 is conventionally interpreted as end of file.

Write writes *nbytes* bytes of data starting at *buf* to the file associated with *fd* at the file offset. The offset is advanced by the number of bytes written. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

SEE ALSO

intro(2), *open*(2), *dup*(2), *pipe*(2)

DIAGNOSTICS

These functions set *errstr.*

NAME

regcomp, regcomplit, regcompnl, regexec, regsub, rregexec, rregsub, regerror − regular expression

SYNOPSIS

```
#include <regexp.h>

Reprog  *regcomp(char *exp)

Reprog  *regcomplit(char *exp)

Reprog  *regcompnl(char *exp)

int     regexec(Reprog *prog, char *string, Resub *match, int msize)

void    regsub(char *source, char *dest, Resub *match, int msize)

int     rregexec(Reprog *prog, Rune *string, Resub *match, int msize)

void    rregsub(Rune *source, Rune *dest, Resub *match, int msize)

void    regerror(char *msg)
```

DESCRIPTION

Regcomp compiles a regular expression and returns a pointer to the generated description. The space is allocated by *malloc*(2) and may be released by *free.* Regular expressions are exactly as in *regexp*(6).

Regcomplit is like *regcomp* except that all characters are treated literally. *Regcompnl* is like *regcomp* except that the `.` metacharacter matches all characters, including newlines.

Regexec matches a null-terminated *string* against the compiled regular expression in *prog.* If it matches, *regexec* returns `1` and fills in the array *match* with character pointers to the substrings of *string* that correspond to the parenthesized subexpressions of *exp*: `match[i].sp` points to the beginning and `match[i].ep` points just beyond the end of the *i*th substring. (Subexpression *i* begins at the *i*th left parenthesis, counting from 1.) Pointers in `match[0]` pick out the substring that corresponds to the whole regular expression. Unused elements of *match* are filled with zeros. Matches involving `*`, `+`, and `?` are extended as far as possible. The number of array elements in *match* is given by *msize.* The structure of elements of *match* is:

```
typedef struct {
        union {
           char *sp;
           Rune *rsp;
        };
        union {
           char *ep;
           Rune *rep;
        };
} Resub;
```

If `match[0].sp` is nonzero on entry, *regexec* starts matching at that point within *string*. If `match[0].ep` is nonzero on entry, the last character matched is the one preceding that point.

Regsub places in *dest* a substitution instance of *source* in the context of the last *regexec* performed using *match.* Each instance of `\n`, where *n* is a digit, is replaced by the string delimited by `match[n].sp` and `match[n].ep`. Each instance of `&` is replaced by the string delimited by `match[0].sp` and `match[0].ep`.

Regerror, called whenever an error is detected in *regcomp, regexec,* or *regsub,* writes the string *msg* on the standard error file and exits. *Regerror* can be replaced to perform special error processing.

Rregexp and *rregsub* are variants of *regexec* and *regsub* that use strings of `Runes` instead of strings of `chars`. With these routines, the *rsp* and *rep* fields of the *match* array elements should be used.

SEE ALSO

grep(1)

DIAGNOSTICS

Regcomp returns `0` for an illegal expression or other failure. *Regexec* returns 0 if *string* is not matched.

NAME

remove − remove a file

SYNOPSIS

```
int remove(char *file)
```

DESCRIPTION

Remove removes *file* from the directory containing it and discards the contents of the file. The user must have write permission in the containing directory. If *file* is a directory, it must be empty.

SEE ALSO

intro(2), *remove*(5)

DIAGNOSTICS

Sets *errstr.*

NAME

rendezvous − user level process synchronization

SYNOPSIS

```
ulong rendezvous(ulong tag, ulong value)
```

DESCRIPTION

The rendezvous system call allows two processes to synchronize and exchange a value. In conjunction with the shared memory system calls (see *segattach*(2) and *fork*(2)), it enables parallel programs to access the system scheduler.

Two processes wishing to synchronize call *rendezvous* with a common *tag*, typically an address in memory they share. One process will arrive at the rendezvous first; it suspends execution until a second arrives. When a second process meets the rendezvous the *value* arguments are exchanged between the processes and returned as the result of the respective *rendezvous* system calls. Both processes are awakened when the rendezvous succeeds.

The tag space is common to processes in the same file name space.

If a rendezvous is interrupted the return value is `~0`, so that value should not be used in normal communication.

SEE ALSO

segattach(2), *fork*(2)

DIAGNOSTICS

Sets *errstr.*

NAME

RGB, rgbpix, rdcolmap, wrcolmap − handle color screens

SYNOPSIS

```
ulong rgbpix(Bitmap *b, RGB rgb)

void  rdcolmap(Bitmap *b, RGB *map)

void  wrcolmap(Bitmap *b, RGB *map)
```

DESCRIPTION

Colors are described by the red, green, and blue light intensities, in an `RGB` datum:

```
typedef
struct RGB {
     ulong red;
     ulong green;
     ulong blue;
} RGB;
```

Black is represented by zero in all three positions and white has the maximum `unsigned long` value in all three positions.

Some of the graphics functions, such as *point* (see *bitblt*(2)), take a *pixel value* argument, which is a single `unsigned long`. For a given bitmap, *rgbpix* returns the pixel value with a color closest to the color represented by the *rgb* argument.

There is a *colormap* associated with each Bitmap. A colormap is an array of `RGB`s, of length $2^{2^{ldepth}}$, giving the colors for pixels 0, 1, 2, etc.

Rdcolmap reads the colormap for the given bitmap into the provided *map*, which must have enough space to hold it. *Wrcolmap* associates the given colormap with the given bitmap, if possible. (The hardware might not allow this.)

BUGS

These functions work only for the screen bitmap. This interface will have to be refined for screens with more than 8 bits per pixel.

SEE ALSO

graphics(2)

NAME

runetochar, chartorune, runelen, fullrune, utflen, utfrune, utfrrune, utfutf − rune/UTF conversion

SYNOPSIS

```
int   runetochar(char *s, Rune *r)

int   chartorune(Rune *r, char *s)

int   runelen(long r)

int   fullrune(char *s, int n)

int   utflen(char *s)

char* utfrune(char *s, long c)

char* utfrrune(char *s, long c)

char* utfutf(char *s1, char *s2)
```

DESCRIPTION

These routines convert to and from a UTF byte stream and runes.

Runetochar copies one rune at *r* to at most `UTFmax` characters starting at *s* and returns the number of characters copied. `UTFmax`, defined as `3` in `<libc.h>`, is the maximum number of bytes required to represent a rune.

Chartorune copies at most `UTFmax` characters starting at *s* to one rune at *r* and returns the number of characters copied. If the characters are not exactly in UTF format, *chartorune* will convert to 0x80 and return 1.

Runelen returns the number of characters required to convert *r* into UTF.

Fullrune returns 1 if the string *s* of length *n* is long enough to be decoded by *chartorune* and 0 otherwise. This does not guarantee that the string contains a legal UTF encoding. This routine is used by programs that obtain input a character at a time and need to know when a full rune has arrived.

The following routines are analogous to the corresponding string routines with `utf` substituted for `str` and `rune` substituted for `chr`.

Utflen returns the number of runes that are represented by the UTF string *s*.

Utfrune (*utfrrune*) returns a pointer to the first (last) occurrence of rune *c* in the UTF string *s*, or 0 if *c* does not occur in the string. The NUL character terminating a string is considered to be part of the string *s*.

Utfutf returns a pointer to the first occurrence of the UTF string *s2* as a UTF substring of *s1*, or 0 if there is none. If *s2* is the null string, *utfutf* returns *s1*.

SEE ALSO

utf(6), *tcs*(1),

NAME

seek − change file offset

SYNOPSIS

```
long seek(int fd, long n, int type)
```

DESCRIPTION

Seek sets the offset for the file associated with *fd* as follows:

> If *type* is 0, the offset is set to *n* bytes.
>
> If *type* is 1, the pointer is set to its current location plus *n*.
>
> If *type* is 2, the pointer is set to the size of the file plus *n*.

The new file offset value is returned.

Seeking far beyond the end of a file, then writing, creates a gap, or ‘hole,’ that occupies no physical space and reads as zeros.

Seeking in a directory is not allowed.

SEE ALSO

intro(2), *open*(2)

DIAGNOSTICS

Sets *errstr.*

NAME

segattach, segdetach, segfree − map/unmap a segment in virtual memory

SYNOPSIS

```
int   segattach(int attr, char *class, void *va, ulong len)

int   segdetach(void *addr)

int   segfree(void *va, ulong len)
```

DESCRIPTION

Segattach creates a new memory segment and adds it to the calling process's address space. Segments are identified by system-dependent classes. Segment classes `memory` (plain memory) and `shared` (shared memory) should be available on all systems.

Shared segments are inherited by the children of the attaching process and remain untouched across a *fork*(2). An *exec*(2) will release a shared segment if it overlaps the segments in the file being *exec'ed*; otherwise the segment will be inherited.

Some machines provide a segment class `lock`. Lock segments allow access to special lock hardware provided by some multiprocessors, in particular the SGI Power Series machines.

Systems may also provide interfaces to special hardware devices like frame buffers through the *segattach* interface. Device memory mapped by this method is typically uncached by default.

If the specified *class* is unknown, *segattach* draws an error.

Attr specifies the new segment's attributes. The only attribute implemented on all classes of segment is `SG_RONLY`, which allows only read access on the segment. Specific devices may implement attributes to control caching and allocation, but these will vary between devices.

Va and *len* specify the position of the segment in the process's address space. *Va* is rounded down to the nearest page boundary and *va+len* is rounded up. The system does not permit segments to overlap.

Segdetach removes a segment from a process's address space. Memory used by the segment is freed. *Addr* may be any address within the bounds of the segment.

The system will not permit the text and stack segments to be detached from the address space.

Segfree allows specific areas of a segment's memory to be freed. *Va* and *len* are interpreted as in *segattach* but need not refer to the entire segment.

To select a virtual address to which a segment can be attached, the following algorithm is reliable. Read the `segment` file of the current process (see *proc*(3)) to find the base of the stack segment. Subtract the size of the new segment and use that address.

The MIPS R2000 and R3000 have no hardware instructions to implement locks. The following method can be used to build them from software. First, try to *segattach* a segment of class `lock`. If this succeeds, the machine is an SGI Power Series and the memory contains hardware locks. Each 4096-byte page has 64 `long` words at its beginning; each word implements a test-and-set semaphore when read; the low bit of the word is zero on success, one on failure. If the *segattach* fails, there is no hardware support but the operating system helps: Any `COP3` instruction will be trapped by the kernel and interpreted as a test-and-set. In the trap, `R1` points to a `long`; on return, `R1` is greater or equal zero on success, negative on failure. The following assembly language implements such a test-and-set.

```
/*
 *     MIPS test and set
 */
       TEXT   tas(SB), $0
btas:
       MOVW   sema+0(FP), R1
       MOVB   R0, 1(R1)
       NOR    R0, R0, R0    /* NOP */
```

```
                    WORD   $(023<<26)    /* MFC3 R0, R0 */
                    BLTZ   R1, btas
                    RET
```

SEE ALSO

segbrk(2), *segflush*(2)
`/proc/*/segment`

DIAGNOSTICS

These functions set *errstr*.

NAME

segbrk − change memory allocation

SYNOPSIS

```
int   segbrk(void *saddr, void *addr)
```

DESCRIPTION

Segbrk sets the system's idea of the lowest unused location of a segment to *addr* rounded up to the next multiple of 4 bytes. The segment is identified by *saddr* which may be any valid address within the segment.

A call to *segbrk* with a zero *addr* argument returns the address of the top of bss.

The system will prevent segments from overlapping and will not allow the text and data segment lengths to be altered.

SEE ALSO

segattach(2), *segflush*(2)
`/proc/*/segment`

DIAGNOSTICS

Sets *errstr.*

NAME

segflush - flush segment memory cache

SYNOPSIS

```
int   segflush(void *va, ulong len)
```

DESCRIPTION

Segflush flushes the instruction cache associated with pages contained in a segment. All subsequent new pages in the segment will also be flushed when first referenced.

Va is an address within the segment to be flushed; it is rounded down to the nearest page boundary. *Len* specifies the length in bytes of the memory to flush; *va+len* is rounded up to the nearest page boundary.

SEE ALSO

segattach(2), *segbrk*(2)
`/proc/*/segment`

DIAGNOSTICS

Sets *errstr.*

NAME

setjmp, longjmp, notejmp − non-local goto

SYNOPSIS

```
int  setjmp(jmp_buf env)

void longjmp(jmp_buf env, int val)

void notejmp(void *uregs, jump_buf env, int val)
```

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

Longjmp restores the environment saved by the last call of *setjmp*. It then causes execution to continue as if the call of *setjmp* had just returned with value *val*. The invoker of *setjmp* must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called.

Notejmp is the same as *longjmp* except that it is to be called from within a note handler (see *notify*(2)). The *uregs* argument should be the first argument passed to the note handler.

Setjmp and *longjmp* can also be used to switch stacks. Defined in `<u.h>` are several macros that can be used to build `jmp_bufs` by hand. The following code establishes a `jmp_buf` that may be called by *longjmp* to begin execution in a function `f` with 1024 bytes of stack:

```
#include <u.h>
#include <libc.h>

jmp_buf label;
#define NSTACK 1024
char stack[NSTACK];

void
setlabel(void)
{
     label[JMPBUFPC] = ((ulong)f+JMPBUFDPC);
     /* -2 leaves room for old pc and new pc in frame */
     label[JMPBUFSP] = (ulong)(&stack[NSTACK-2*sizeof(ulong*)]);
}
```

BUGS

Notejmp cannot recover from an address trap or bus error (page fault) on the 680x0 architectures.

SEE ALSO

notify(2)

NAME

sin, cos, tan, asin, acos, atan, atan2 − trigonometric functions

SYNOPSIS

```
double sin(double x)

double cos(double x)

double tan(double x)

double asin(double x)

double acos(double x)

double atan(double x)

double atan2(double y, double x)
```

DESCRIPTION

Sin, cos and *tan* return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sine in the range $-\pi/2$ to $\pi/2$.

Acos returns the arc cosine in the range 0 to π.

Atan returns the arc tangent in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arc tangent of *y/x* in the range $-\pi$ to π.

SEE ALSO

intro(2)

BUGS

The value of *tan* for arguments greater than about 2^{31} is garbage.

NAME

sinh, cosh, tanh − hyperbolic functions

SYNOPSIS

```
double sinh(double x)

double cosh(double x)

double tanh(double x)
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

SEE ALSO

intro(2)

NAME

sleep, alarm − delay, ask for delayed note

SYNOPSIS

```
int sleep(long millisecs)

long alarm(unsigned long millisecs)
```

DESCRIPTION

Sleep suspends the current process for the number of milliseconds specified by the argument. The actual suspension time may be a little more or less than the requested time. A sleep of 0 causes the process to give up the CPU if another process is ready to run. Sleep returns −1 if interrupted, 0 otherwise.

Alarm causes an `alarm` note (see *notify*(2)) to be sent to the invoking process after the number of milliseconds given by the argument. Successive calls to *alarm* reset the alarm clock. A zero argument clears the alarm. The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

intro(2)

DIAGNOSTICS

These functions set *errstr.*

NAME

stat, fstat, wstat, fwstat, dirstat, dirfstat, dirwstat, dirfwstat − get and put file status

SYNOPSIS

```
int stat(char *name, char *edir)

int fstat(int fd, char *edir)

int wstat(char *name, char *edir)

int fwstat(int fd, char *edir)

int dirstat(char *name, Dir *dir)

int dirfstat(int fd, Dir *dir)

int dirwstat(char *name, Dir *dir)

int dirfwstat(int fd, Dir *dir)
```

DESCRIPTION

Given a file's *name*, or an open file descriptor *fd*, these routines retrieve or modify file status information. *Stat*, *fstat*, *wstat*, and *fwstat* are the system calls; they deal with machine-independent *directory entries*. Their format is defined by *stat*(5). *Stat* and *fstat* retrieve information about *name* or *fd* into *edir*, a buffer of length `DIRLEN`, defined in `<libc.h>`. *Wstat* and *fwstat* write information back, thus changing file attributes according to *edir*.

Dirstat, *dirfstat*, *dirwstat*, and *dirfwstat* are the same as their counterparts, except that they operate on *Dir* structures:

```
typedef
struct Dir {
     char    name[NAMELEN];    /* last element of path */
     char    uid[NAMELEN];     /* owner name */
     char    gid[NAMELEN];     /* group name */
     Qid     qid;              /* unique id from server */
     long    mode;             /* permissions */
     long    atime;            /* last read time */
     long    mtime;            /* last write time */
     Length;                   /* file length: see <u.h> */
     short   type;             /* server type */
     short   dev;              /* server subtype */
} Dir;
```

This structure, the *Qid* structure, `NAMELEN`, and `DIRLEN` are defined in `<libc.h>.` The *Length* structure is defined in `<u.h>`. *Length* is an unnamed structure (see *2c*(1)), which means that its fields are directly accessible; if the length is known to fit in a `long`, then use `length` as a field name to retrieve it. If the file resides on permanent storage and is not a directory, the length returned by *stat* is the number of bytes in the file. For directories, the length returned is zero. For files that are streams (e.g., pipes and network connections), the length is the number of bytes that can be read without blocking.

Each file is the responsibility of some *server*: it could be a file server, a kernel device, or a user process. `Type` identifies the server type, and `dev` says which of a group of servers of the same type is the one responsible for this file. `Qid` is a structure containing *path* and *vers* fields, each an `unsigned long`: *path* is guaranteed to be unique among all path names currently on the file server, and *vers* changes each time the file is modified. Thus, if two files have the same `type`, `dev`, and `qid` they are the same file.

The bits in `mode` are defined by

`0x80000000`	directory
`0x40000000`	append only
`0x20000000`	exclusive use (locked)

`0400`	read permission by owner
`0200`	write permission by owner
`0100`	execute permission (search on directory) by owner
`0070`	read, write, execute (search) by group
`0007`	read, write, execute (search) by others

There are constants defined in `<libc.h>` for these bits: `CHDIR`, `CHAPPEND`, and `CHEXCL` for the first three; and `CHREAD`, `CHWRITE`, and `CHEXEC` for the read, write, and execute bits for others.

The two time fields are measured in seconds since the epoch (Jan 1 00:00 1970 local time). `Mtime` is the time of the last change of content. Similarly, `atime` is set whenever the contents are accessed; also, it is set whenever *mtime* is set.

`Uid` and `gid` are the names of the owner and group of the file. Groups are also users, but each server is free to associate a list of users with any user name *g*, and that list is the set of users in the group *g*. When an initial attachment is made to a server, the user string in the process group is communicated to the server. Thus, the server knows, for any given file access, whether the accessing process is the owner or in the group of the file. This selects which sets of three bits in `mode` is used to check permissions.

Only some of the fields may be changed with the *wstat* calls. The *name* can be changed by anyone with write permission in the parent directory. The *mode* can be changed by the owner or the group leader of the file's current group. The *gid* can be changed by the owner if he or she is a member of the new group. The *gid* can be changed by the group leader of the file's current group if he or she is the leader of the new group. (See *intro*(5) for permission information, and *users*(6) for user and group information).

SEE ALSO

intro(2), *fcall*(2), *dirread*(2), *stat*(5)

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strdup, strstr − string operations

SYNOPSIS

```
char* strcat(char *s1, char *s2)

char* strncat(char *s1, char *s2, long n)

int   strcmp(char *s1, char *s2)

int   strncmp(char *s1, char *s2, long n)

char* strcpy(char *s1, char *s2)

char* strncpy(char *s1, char *s2, long n)

long  strlen(char *s)

char* strchr(char *s, char c)

char* strrchr(char *s, char c)

char* strpbrk(char *s1, char *s2)

long  strspn(char *s1, char *s2)

long  strcspn(char *s1, char *s2)

char* strtok(char *s1, char *s2)

char* strdup(char *s)

char* strstr(char *s1, char *s2)
```

DESCRIPTION

The arguments *s1, s2* and *s* point to null-terminated strings. The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* bytes. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*. *Strncmp* makes the same comparison but examines at most *n* bytes. The comparisons are made with unsigned bytes.

Strcpy copies string *s2* to *s1*, stopping after the null byte has been copied. *Strncpy* copies exactly *n* bytes, truncating *s2* or adding null bytes to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Strlen returns the number of bytes in *s*, not including the terminating null byte.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of byte *c* in string *s*, or `0` if *c* does not occur in the string. The null byte terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any byte from string *s2*, `0` if no byte from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of bytes from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more bytes from the separator string *s2*. The first call, with pointer *s1* specified, returns a pointer to the first byte of the first token, and will have written a null byte into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls; subsequent calls, signified by *s1* being `0`, will work through the string *s1* immediately following that token. The separator string *s2* may be different from call to call. When no token remains in *s1*, `0` is returned.

Strdup returns a pointer to a distinct copy of the null-terminated string *s* in space obtained from *malloc*(2) or `0` if no space can be obtained.

Strstr returns a pointer to the first occurrence of *s2* as a substring of *s1*, or 0 if there is none. If *s2* is the null string, *strstr* returns *s1*.

SEE ALSO

memory(2), *rune*(2)

BUGS

These routines know nothing about UTF. Use the routines in *rune*(2) as appropriate.

The outcome of overlapping moves varies among implementations.

NAME

subfalloc, subffree, rdsubfontfile, wrsubfontfile, mkfont − subfont manipulation

SYNOPSIS

```
#include <u.h>
#include <libc.h>
#include <libg.h>

Subfont* subfalloc(int n, int height, int ascent,
         Fontchar *info, Bitmap *b, ulong q0, ulong q1)

void     subffree(Subfont *f)

Subfont* rdsubfontfile(int fd, Bitmap *b)

void     wrsubfontfile(int fd, Subfont *f)

Font*    mkfont(Subfont *f);
```

DESCRIPTION

Subfonts are the components of fonts that hold the character images. A font is composed from an array of subfonts; see *cachechars*(2). A new `Subfont` is allocated and initialized with *subfalloc*. See *cachechars*(2) for the meaning of *n*, *height*, *ascent*, and *info*, and the arrangement of characters in bitmap *b*. The fields of the returned `Subfont` structure are set to the passed arguments, and the `id` field is set to the identifying number used by `/dev/bitblt` (see *bit*(3)). After a successful *subfalloc*, *b* is attached to the subfont and is unavailable to the application; it should not be used. *Subfalloc* returns 0 on failure.

The *q0* and *q1* arguments are used as tags in a cache of subfonts (see below). If all ones, they disable caching.

Subffree frees a subfont and all its associated structure including the associated bitmap. Since *subffree* calls *free* on `f->info`, if `f->info` was not allocated by *malloc*(2) it should be zeroed before calling *subffree.*

A number of subfonts are kept in external files. The convention for naming subfont files is:

> `/lib/font/bit/`*name*`/`*class*`.`*size*`.`*ldepth*

where *size* is approximately the height in pixels of the lower case letters (without ascenders or descenders). If there is only one version of the subfont, the `.`*ldepth* extension is elided. *Class* describes the range of runes encoded in the subfont: `ascii`, `latin1`, `greek`, etc.

The format of a subfont file is described in *font*(6)). Briefly, it contains a bitmap with all the characters in it, followed by a subfont header, followed by character information. *Rdsubfontfile* reads a subfont from the file descriptor *fd.* If *b* is zero, *rdsubfontfile* reads the bitmap as well as the character information from the file and allows the resulting subfont and bitmap to be cached in the server for sharing with other applications. The first thing such an *rdsubfontfile* call does is check to see if the subfont can be recovered directly from the server: if the qid (see *intro*(5)) of the file pointed to by *fd* matches `q0` and `q1` of a cache entry, the subfont is read from the server instead of from *fd.* This is the normal use of *rdsubfontfile.*

Unusual applications such as font editors may choose to have uncached (hence unshared) subfonts or to associate arbitrary bitmaps with the character data. If the *b* argument to *rdsubfontfile* is non-null, *rdsubfontfile* reads only the character information from *fd* (which must be positioned after the bitmap in the file) and the resulting subfont is uncached. Nonetheless, *rdsubfontfile* calls *subfalloc* with *b* and hence *b* is unusable afterwards; make a copy first if necessary. *Rdsubfontfile* returns 0 on error.

Wrsubfontfile writes on *fd* the part of a subfont file that comes after the bitmap. Because the bitmap of a cached subfont is unavailable to the application, without extraordinary measures only uncached subfonts may be written whole to files.

Mkfont takes as argument a `Subfont` *s* and returns a pointer to a `Font` that maps the character images in *s* into the `Runes` *min* to *min*+*s*`->n-1`.

FILES

`/lib/font/bit` bitmap font file tree

SEE ALSO

graphics(2), *balloc*(2), *bitblt*(2), *cachechars*(2), *bitmap*(6), *font*(6)

DIAGNOSTICS

All of the functions use the graphics error function (see *graphics*(2)).

BUGS

Subfonts can contain no more than about 1300 characters.

NAME

syminit, getsym, symbase, pc2sp, pc2line, line2addr, lookup, findlocal, getauto, findsym, localsym, globalsym, textsym, file2pc, fileelem, filesym, fileline, symerror − symbol table access functions

SYNOPSIS

```
#include <bio.h>
#include <mach.h>

int  syminit(int fd, Fhdr *fp)

Sym  *getsym(int index)

Sym  *symbase(long *nsyms)

int  fileelem(Sym **fp, uchar *encname, char *buf, int n)

int  filesym(int index, char *buf, int n)

long pc2sp(ulong pc)

long pc2line(ulong pc)

long line2addr(ulong line, ulong basepc)

int  lookup(char *fn, char *var, Symbol *s)

int  findlocal(Symbol *s1, char *name, Symbol *s2)

int  getauto(Symbol *s1, int off, int class, Symbol *s2)

int  findsym(long addr, int class, Symbol *s)

int  localsym(Symbol *s, int index)

int  globalsym(Symbol *s, int index)

int  textsym(Symbol *s, int index)

long file2pc(char *file, ulong line)

int fileline(char *str, int n, ulong addr)

Map  *newmap(Map *map, int fd)

extern char *symerror
```

DESCRIPTION

These functions provide machine-independent access to the symbol tables of an executable file or executing process image. The latter is accessible by opening the device `/proc/`*pid*`/text` as described in *proc*(3). The functions are stored in library `libmach.a`; the library is automatically searched by the loader when header file *mach.h* is included in a source file. *Mach*(2) and *object*(2) describe additional library functions for processing symbol tables and object files.

Syminit, *getsym*, *symbase*, *fileelem*, *pc2sp*, *pc2line*, and *line2addr* process the symbol table contained in an executable file or the `text` file associated with an executing program. The symbol table is stored internally as an array of `Sym` data structures as defined in *a.out*(6).

Syminit uses the data in the `Fhdr` structure filled by *crackhdr* (see *mach*(2)) to read the raw symbol tables from the open file descriptor *fd*. It returns the count of the number of symbols or -1 if an error occurs.

Getsym returns the address of the *i*th `Sym` structure or zero if *index* is out of range.

Symbase returns the address of the first `Sym` structure in the symbol table. The number of entries in the symbol table is returned in *nsyms* .

Fileelem converts a file name, encoded as described in *a.out*(6), to a character string. *Fp* is the base of an array of pointers to file path components ordered by path index. *Encname* is the address of an array of encoded file path components in the form of a `z` symbol table entry. *Buf* and *n* specify the address of a receiving character buffer and its length. *Fileelem* returns the length of the null-terminated string that is at most *n-1* bytes long.

Filesym is a higher-level interface to *fileelem*. It fills *buf* with the name of the *i*th file and returns the length of the null-terminated string that is at most *n-1* bytes long. The file names are retrieved in no particular order, although the order of retrieval does not vary from one pass to the next. A zero is returned when *index* is too large or too small or an error occurs during file name conversion.

Pc2sp returns an offset associated with a given value of the program counter. Adding this offset to the current value of the stack pointer gives the address of the current stack frame. This approach only applys to the 386 and 68020 architectures; other architectures use a fixed stack frame accessible through a dummy local variable defined in the symbol table.

Pc2line returns the line number of the statement associated with the instruction address *pc*. The line number is the absolute line number in the file as seen by the compiler after pre-processing; the original line number in the source file may derived from this value using the history stacks contained in the symbol table.

Line2addr converts a line number to an instruction address. The first argument is the absolute line number in a file. Since a line number does not uniquely identify an instruction location (every source file has line 1), a second argument specifies a text address from which the search begins. Usually this is the address of the first function in the file of interest.

Pc2sp, *pc2line*, and *line2addr* return −1 in the case of an error.

Lookup, *findlocal*, *getauto*, *findsym*, *localsym*, *globalsym*, *textsym*, *file2pc*, and *fileline* operate on data structures riding above the raw symbol table. These data structures occupy memory and impose a startup penalty but speed retrievals and provide higher-level access to the basic symbol table data. *Syminit* must be called prior to invoking these functions. The `Symbol` data structure:

```
typedef struct {
        void *handle;     /* private */
        struct {
            char  *name;
            long   value;
            char   type;
            char   class;
        };
} Symbol;
```

describes a symbol table entry. The `value` field contains the offset of the symbol within its address space: global variables relative to the beginning of the data segment, text beyond the start of the text segment, and automatic variables and parameters relative to the stack frame. The `type` field contains the type of the symbol as defined in *a.out*(6). The `class` field assigns the symbol to a general class; `CTEXT`, `CDATA`, `CAUTO`, and `CPARAM` are the most popular.

Lookup fills a `Symbol` structure with symbol table information. Global variables and functions are represented by a single name; local variables and parameters are uniquely specified by a function and variable name pair. Arguments *fn* and *var* contain the name of a function and variable, respectively. If both are non-zero, the symbol table is searched for a parameter or automatic variable. If only *var* is zero, the text symbol table is searched for function *fn*. If only *fn* is zero, the global variable table is searched for *var*.

Findlocal fills *s2* with the symbol table data of the automatic variable or parameter matching *name*. *S1* is a `Symbol` data structure describing a function or a local variable; the latter resolves to its owning function.

Getauto searches the local symbols associated with function *s1* for an automatic variable or parameter located at stack offset *off*. *Class* selects the class of variable: `CAUTO` or `CPARAM`. *S2* is the address of a `Symbol` data structure to receive the symbol table information of the desired symbol.

Findsym returns the symbol table entry of type *class* stored near *addr*. The selected symbol is a global variable or function with address nearest to and less than or equal to *addr*. Class specification `CDATA` searches only the global variable symbol table; class `CTEXT` limits the search to the text symbol table.

Class specification `CANY` searches the text table first, then the global table.

Localsym returns the *i*th local variable associated with the function indicated by *s*. *S* may reference a function or a local variable; the latter resolves to its owning function. If the *i*th local symbol exists, *s* is filled with the data describing it.

Globalsym loads *s* with the symbol table information of the *i*th global variable.

Textsym loads *s* with the symbol table information of the *i*th text symbol. The text symbols are ordered by increasing address.

File2pc returns a text address associated with *line* in file *file*.

Fileline converts text address *addr* to its equivalent line number in a source file. The result, a null terminated character string of the form `file:line` is placed in buffer *str* of *n* bytes. Up to *n-1* characters are copied to the buffer.

Functions *file2pc* and *fileline* may produce inaccurate results when applied to optimized code.

Unless otherwise specified, all functions return 1 on success, or 0 on error.

SEE ALSO

mach(2), *object*(2), *proc*(3), *a.out*(6)

NAME

time − time in seconds since epoch

SYNOPSIS

```
long time(long *tp)
```

DESCRIPTION

Time returns the number of seconds since the epoch 00:00:00 GMT, Jan. 1, 1970. If *tp* is not zero then `*`*tp* is also set to the answer.

This function works by reading `/dev/time`, opening that file when *time* is first called.

SEE ALSO

cons(3)

DIAGNOSTICS

Sets *errstr.*

NAME

tmpfile, tmpnam − stdio temporary files

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void)

char *tmpnam(char *s)
```

DESCRIPTION

Tmpfile creates a temporary file that will automatically be removed when the file is closed or the program exits. The return value is a stdio `FILE*` opened in update mode (see *fopen*(2)).

Tmpnam generates a string that is a valid file name and that is not the same as the name of an existing file. If *s* is zero, it returns a pointer to a string which may be overwritten by subsequent calls to *tmpnam*. If *s* is non-zero, it should point to an array of at least `L_tmpnam` (defined in `<stdio.h>`) characters, and the answer will be copied there.

FILES

`/tmp/tf000000000000`	template for *tmpfile* file names.
`/tmp/tn000000000000`	template for *tmpnam* file names.

BUGS

The files created by *tmpfile* are not removed until *exits*(2) is executed; in particular, they are not removed on *fclose* or if the program terminates abnormally.

NAME

wait − wait for a process to exit

SYNOPSIS

`int wait(Waitmsg *w)`

DESCRIPTION

Wait causes a process to wait for any child process (see *fork*(2)) to exit. It returns the pid of a child that has exited and fills in *w* with more information about the child. *W* points to a `Waitmsg`, which has this structure:

```
typedef
struct Waitmsg
{
     char pid[12];          /* of loved one */
     char time[3*12];       /* of loved one & descendants */
     char msg[ERRLEN];
} Waitmsg;
```

`Pid` is the child's `pid`. The `time` array contains the time the child and its descendants spent in user code, the time spent in system calls, and the child's elapsed real time, all in units of milliseconds. All integers in a `Waitmsg` are formatted as right-justified textual numbers in 11-byte fields followed by a blank. `Msg` contains the message that the child specified in *exits*(2). For a normal exit, `msg[0]` is zero, otherwise `msg` is prefixed by the process name, a blank, the process id, and a colon.

If there are no more children to wait for, *wait* returns immediately, with return value −1.

SEE ALSO

fork(2), *exits*(2)

DIAGNOSTICS

Sets *errstr.*

NAME

intro − introduction to the Plan 9 devices

DESCRIPTION

A Plan 9 *device* implements a file tree for client processes. A file name beginning with a pound sign, such as `#c`, names the root of a file tree implemented by a particular *kernel device driver* identified by the character after the pound sign. Such names are usually bound to conventional locations in the name space. For example, after

```
bind("#c", "/dev", MREPL)
```

an *ls*(1) of `/dev` will list the files provided by the *console* device.

A kernel device driver is a *server* in the sense of the Plan 9 File Protocol, 9P (see Section 5), but with the messages implemented by local rather than remote procedure calls. Also, several of the messages (*Nop*, *Session*, *Flush*, and *Error*) have no subroutine equivalents.

When a system call is passed a file name beginning with `#` it looks at the next character, and if that is a valid *device character* it performs an *attach*(5) on the corresponding device to get a channel representing the root of that device's file tree. If there are any characters after the device character but before the next `/` or end of string, those characters are passed as parameter *aname* to the attach. For example,

```
#Itcp
```

identifies the implementation of the TCP protocol supplied by the IP device (see *ip*(3)).

Each kernel device has a conventional place at which to be bound to the name space. The *SYNOPSIS* sections of the following pages includes a shell *bind* command to put the device in the conventional place. Most of these binds are done automatically by *init*(8).

SEE ALSO

intro(5), *intro*(2)

NAME

arp − Internet Address Resolution Protocol

SYNOPSIS

```
bind −a #a /net/arp

/net/arp/ctl
/net/arp/data
/net/arp/stats
```

DESCRIPTION

The *arp* device provides the means by which the kernel resolves IP addresses into Ethernet addresses. A cache is maintained by the `arp` device to speed the process.

The `ctl` file controls the ARP cache maintained by the kernel. The `flush` control message invalidates all entries in the cache. The `delete` *ipaddr* control message invalidates a single cache entry.

The `data` file provides two interfaces. The first open of the `data` file connects the `arpd` server to the kernel ARP cache (see *ipconfig*(8)). `Arpd` writes the results of address resolution requests from the kernel back into the `data` file to prime the cache. Subsequent opens of the `data` file allow the contents of the cache to be read. Each cache entry consists of an IP address, an Ethernet address and the status of the entry. Entries may be invalid, permanent or temporary. Permanent entries will never be aged from the cache. Temporary entries may be replaced by new addresses entered by the ARP server.

The file `stats` reports the cache performance.

SEE ALSO

ip(3), *ipconfig*(8)

NAME

async − framing for a serial line to Datakit

SYNOPSIS

```
Fctlfd = open(".../ctl", ORDWR);
Fwrite(ctlfd, "push async", 10);
```

DESCRIPTION

This is not a device, but rather a *stream module* (see *stream*(3)) that can be pushed onto a stream. This module provides the framing necessary to treat a serial line as a Datakit trunk. It is usually pushed onto a stream before the *dkmux* module. The frame includes a CRC. Any received frames with an incorrect CRC are discarded.

The format of a message upstream of the module is:

channel # low byte
channel # high byte
control byte (0 means none)
data bytes

The format of a frame is:

`0x7d`
`0x7d`
channel # low byte
channel # high byte
crc low byte
crc high byte
`0x7d`
`0x7d`

All control bytes in the frame are preceded by a `0x9d` byte. All `0x9d` and `0x7d` bytes in the data are followed by a `0x00` byte to distinguish them from framing or control specifiers.

SEE ALSO

stream(3), *cons*(3), *dk*(3)

NAME

bit − screen graphics, mouse

SYNOPSIS

```
bind #b /dev

/dev/bitblt
/dev/mouse
/dev/screen

#include <u.h>
#include <libg.h>

ushortBGSHORT(uchar *p)
ulong BGLONG(uchar *p)
void  BPSHORT(uchar *p, ushort v)
void  BPLONG(uchar *p, ulong v)
```

DESCRIPTION

The *bit* device provides the `bitblt`, `mouse`, and `screen` on machines with a bitmapped screen and a mouse. The device is exclusive use.

The *bit* device provides, through the `bitblt` file, access to *bitmaps, fonts,* and *subfonts* in its private storage, as described in *graphics*(2). Each object is identified by a short, its *id.* The bitmap with id zero is special: it represents the visible display. The subfont with id zero is also special: it is initialized to a default subfont that is always available. There is no default font. There is also a cursor associated with the screen; it is always displayed at the current mouse position. A process can write messages to `bitblt` to allocate and free bitmaps, fonts, and subfonts, read or write portions of the bitmaps, and draw line segments, textures, and character strings in the bitmaps. All graphics requests are clipped to their bitmaps. Some messages return a response to be recovered by reading `bitblt`.

The format of messages written to `bitblt` is a single lower case letter followed by binary parameters; multibyte integers are transmitted with the low order byte first. The `BPSHORT` and `BPLONG` macros place correctly formatted two- and four-byte integers into a character buffer. Some messages return a response formatted the same way; it usually starts with the upper case version of the request character. `BGSHORT` and `BGLONG` retrieve values from a character buffer. Points are two four-byte numbers: *x*, *y*. Rectangles are four four-byte numbers: min *x*, min *y*, max *x*, and max *y*.

The following requests are accepted by the `bitblt` file. The numbers in brackets give the length in bytes of the parameters.

`a` *ldepth*[1] *rect*[16]

Allocate a bitmap. *Ldepth* is the log base 2 of the number of bits per pixel. *Rect* is a Rectangle giving the extent of the bitmap. The bitmap is cleared to all zeros. The id of the allocated bitmap is returned on a subsequent *read* from `bitblt`, returning the three bytes: `A` followed by the id.

`b` *dstid*[2] *dstpt*[8] *srcid*[2] *srcrect*[16] *code*[2]

Bit-block transfer (`bitblt`) from a rectangle in the bitmap identified by *srcid* to a congruent rectangle at Point `dstpt` in the bitmap identified by *dstid.* The rectangle is clipped against both source and destination bitmaps. See *bitblt*(2).

`c` [*pt*[8] *clr*[32] *set*[32]]

Switch mouse cursor. See the description of `Cursors` in *graphics*(2) for the meaning of the *pt* (the offset), *set*, and *clr* arguments. If only `c` is provided — that is, if the message is one byte long — the cursor changes to the default, typically an arrow.

`e` *id*[2] *pt*[8] *value*[1] *code*[2] *n*[2] *pts*[*n**2]

Join the *n*+1 points *pt* and *pts* with *n* segments, exactly as for the *l* operator. The *pts* are specified by pairs of signed bytes holding offsets from the previous point in the list.

`f` *id*[2]

Free the resources associated with the allocated bitmap identified by *id.*

`g` *id*[2]

Free the resources associated with the allocated subfont identified by *id,* including its bitmap. If the subfont is cached, the associated data may be recoverable even after it has been freed; see below.

`h` *id*[2]

Free the resources associated with the allocated font identified by *id.*

`i`

Initialize the device. The next operation on `bitblt` should be a *read*(2). A read of length 34 returns information about the display:

`I` *ldepth*[1] *rect*[16] *cliprect*[16].

If the read count is large enough, the above information is followed by the header and character information of the default `Subfont`, in the format expected by *rdsubfontfile* (see *subfalloc*(2) and *font*(6)). 'Large enough' is $36 + 6n$, where n is the number of characters in the font. The ids of the screen bitmap and default subfont are both zero.

`j` *q0*[4] *q1*[4]

Check to see whether a subfont with tags *q0* and *q1* is in the cache. If it is not, the write of the `j` message will draw an error. If it is, the next read of `bitblt` will return

`J` *id*[2]

followed by the subfont information in the same format as returned by an init message; the subfont will then be available for use.

`k` *n*[2] *height*[1] *ascent*[1] *bitmapid*[2] *q0*[4] *q1*[4] *info*[6*(*n*+1)]

Allocate subfont. The parameters are as described in *subfalloc*(2), with *info* in external subfont file format. *Bitmapid* identifies a previously allocated bitmap containing the character images. *Q0* and *q1* are used as labels for the subfont in the cache; if all ones, the subfont will not be cached and hence shared with other applications. The id of the allocated subfont is recovered by reading from `bitblt` the three bytes: `K` followed by the id. Henceforth, the bitmap with id *bitmapid* is unavailable to the application; in effect, it has been freed.

`l` *id*[2] *pt1*[8] *pt2*[8] *value*[1] *code*[2]

Draw a line segment from Point *pt1* to Point *pt2*, using *code* for the drawing function, and *value* as the source pixel. See *segment* in *bitblt*(2). *Id* identifies the destination bitmap.

`m` *id*[2]

Read the colormap associated with the bitmap with the specified *id*. The next read of `bitblt` will return 12×2^n bytes of colormap data where n is the number of bits per pixel in the bitmap.

`n` *height*[1] *ascent*[1] *ldepth*[2] *ncache*[2]

Allocate a font with the given *height, ascent,* and *ldepth.* The id of the allocated font is recovered by reading from `bitblt` the three bytes: `N` followed by the id. The initial cache associated with the font will have *ncache* character entries of zero width.

`p` *id*[2] *pt*[8] *value*[1] *code*[2]

Change the pixel at Point *pt* using *code* for the drawing function, and *value* as the source pixel. See *point* in *bitblt*(2).

`q` *id*[2] *rect*[16]

Set the clipping rectangle for the bitmap with specified *id* to the given rectangle, which will itself be clipped to the bitmap's image rectangle.

`r` *id*[2] *miny*[4] *maxy*[4]

Read rows *ymin*, *ymin*+1, ... *ymax*−1 of the bitmap with the given bitmap id. See the description of *rdbitmap* in *balloc*(2). A subsequent read of `bitblt` will return the requested rows of pixels.

Note: in this case, the response does not begin an `R`, to simplify the reading of large bitmaps.

`s` *id*[2] *pt*[8] *fontid*[2] *code*[2] *n*[2] *indices*[2**n*]
: Draw using code *code* in the bitmap identified by *id* the text string specified by the *n* cache *indices* in font *fontid,* starting with the upper left corner at *pt*.

`t` *dstid*[2] *rect*[16] *srcid*[2] *code*[2]
: Texture the given rectangle in the bitmap identified by *dstid* by overlaying a tiling of the bitmap identified by *srcid* (aligning (0,0) in the two bitmaps), and using *code* as a drawing code for *bitblt*; see *texture* in *bitblt*(2).

`v` *id*[2] *ncache*[2] *width*[2]
: Reset, resize, and clear the cache for font *id;* the maximum width of the *ncache* characters the cache may hold is set to *width.* Must be done before the first load of a cache slot. If the cache cannot be resized, the write of this message will fail but the cache will be unaffected.

`w` *id*[2] *miny*[4] *maxy*[4] *data*[*n*]
: Replace rows *ymin*, *ymin*+1, ... *ymax*−1 of the bitmap with the given bitmap *id* with the values in *data*. See the description of *wrbitmap* in *balloc*(2).

`x` *x*[4] *y*[4]
: Move the cursor so its origin is at (*x*,*y*).

`y` *id*[2] *cacheindex*[2] *subfontid*[2] *subfontindex*[2]
: Load the description and image of character *subfontindex* in subfont *subfontid* into slot *cacheindex* of font *id.*

`z` *id*[2] *map*[*m*]
: Replace the colormap associated with bitmap *id* with *map*, which contains $m=12\times2^n$ bytes of colormap data (see *rgbpix*(2) for the format).

A read of the `mouse` file returns the mouse status: its position and button state. The read blocks until the state has changed since the last read. The read returns 14 bytes:

`m` *buttons*[1] *x*[4] *y*[4] *msec*[4]

where *x* and *y* are the mouse coordinates in the screen bitmap, *msec* is a time stamp, in units of milliseconds, and *buttons* has set the 1, 2, and 4 bits when the mouse's left, middle, and right buttons, respectively, are down.

The `screen` file contains the screen bitmap in the format described in *bitmap*(6).

DIAGNOSTICS

Most messages to `bitblt` can return errors; these can be detected by a system call error on the *write*(see *read*(2)) of the data containing the erroneous message. The most common error is a failure to allocate because of insufficient free resources. Most other errors occur only when the protocol is mishandled by the application.

BUGS

Because each message must fit in a single 9P message, subfonts are limited to about 1300 characters.

NAME

conc − Datakit concentrator

SYNOPSIS

```
ctlfd = open("rawdkdev/ctl", ORDWR);
write(ctlfd, "push conc", 9);
write(ctlfd, "config name nc0 nc1...", n);

bind -a #Kname /dev

dkctlfd = open("/dev/name/n/ctl", ORDWR);
```

DESCRIPTION

The concentrator partitions the channel space of a raw Datakit device (typically `#i` or `#h`) into subspaces of size *nc0*, *nc1*... Channel numbers are adjusted on all messages so that each subdevice sees a channel space starting at zero. This arrangement must agree with the switch configuration as provisioned by the local Datakit operating company.

FILES

`/dev/`*name*`/`*n*`/data`
`/dev/`*name*`/`*n*`/ctl`

SEE ALSO

datakit(3)

NAME

cons − console, clocks, process/process group ids, user, null, klog, stats, lights, noise, sysstat, hz, swap, crypt, chal, key

SYNOPSIS

```
bind #c /dev

/dev/chal
/dev/clock
/dev/cons
/dev/consctl
/dev/cputime
/dev/crypt
/dev/hz
/dev/key
/dev/klog
/dev/lights
/dev/mousectl
/dev/msec
/dev/noise
/dev/null
/dev/pgrpid
/dev/pid
/dev/ppid
/dev/swap
/dev/sysname
/dev/sysstat
/dev/time
/dev/user
```

DESCRIPTION

The console device serves a one-level directory giving access to the console and miscellaneous information.

Reading the `cons` file returns characters typed on the keyboard. Normally, characters are buffered to enable erase and kill processing. A control-U, `^U`, typed at the keyboard *kills* the current input line (removes all characters from the buffer of characters not yet read via `cons`), and a backspace *erases* the previous non-kill, non-erase character from the input buffer. Killing and erasing only delete characters back to, but not including, the last newline. Characters typed at the keyboard actually produce 16-bit runes (see *utf*(6)), but the runes are translated into the variable-length UTF encoding (see *utf*(6)) before putting them into the buffer. A *read*(2) of length greater than zero causes the process to wait until a newline or a `^D` ends the buffer, and then returns as much of the buffer as the argument to `read` allows, but only up to one complete line. A terminating `^D` is not put into the buffer. If part of the line remains, the next `read` will return characters from that remainder and not part of any new line that has been typed since.

If the string `rawon` has been written to the `consctl` file and the file is still open, `cons` is in *raw mode*: characters are not echoed as they are typed, backspace and `^D` are not treated specially, and characters are available to *read* as soon as they are typed. Ordinary mode is reentered when `rawoff` is written to `consctl` or this file is closed.

A *write* (see *read*(2)) to `cons` causes the characters to be printed on the console screen.

The `null` file throws away anything written to it and always returns zero bytes when read.

The `klog` file contains the tail of messages written by kernel logging statements.

Writing a number (as plain text) to the `lights` device directs any lights that are available to turn on and off. The bits of the number are mapped to the lights in a processor-dependent way.

Writing a serial port number (or the string `ps2` for the ps2 port on a pc) configures that port for mouse input.

Writing two blank- or tab- separated numbers to the `noise` device causes the machine to make a tone, if possible. The first number is the frequency, in Hertz, and the second is the duration, in milliseconds.

The `crypt` file performs DES encryption. To encrypt data, first a character `E` (`0x45`) is written to the file, and then the data, which must be at least 8 bytes long. Data longer that 127 bytes is truncated. Data is encrypted with the same algorithm used in *encrypt*(2). The encrypted data can then be read from the file. A similar procedure is used to decrypt data, except an ASCII `D` (`0x44`) is written to the file before the data.

The `key` file is used to set the DES key used for encryption; the key is shared within a process group (see *auth*(6)).

The `chal` file is used for authenticated setting of the user name. When read, it returns an encrypted challenge string to be used for authenticating the user's identity. When written with the appropriate string, the user name and encryption key are set. The format of the strings is documented in *auth*(6).

The rest of the files contain (mostly) read-only strings. Each string has a fixed length: a *read*(2) of more than that gives a result of that fixed length (the result does not include a terminating zero byte); a *read* of less than that length leaves the file offset so the rest of the string (but no more) will be read the next time. To reread the file without closing it, *seek* must be used to reset the offset. When the file contains numeric data, each number is formatted in decimal as an 11-digit number with leading blanks and one trailing blank: twelve bytes total.

The `user` file contains the name of the user associated with the current process.

The `cputime` file holds 6 numbers, containing the time in milliseconds that the current process has spent in user mode, system calls, real elapsed time, and then the time spent, by exited children and their descendants, in user mode, system calls, and real elapsed time.

The `clock` file holds two numbers: the number of clock ticks since booting followed by the number of clock ticks in a second.

The `sysname` file holds the textual name of the machine, e.g. `kremvax`, if known.

The `sysstat` file holds 8 numbers: processor number, context switches, interrupts, system calls, page faults, tlb faults, tlb purges, and load average. If the machine is a multiprocessor, `sysstat` holds one line per processor. Writing anything to `sysstat` resets all of the counts on all processors.

The `swap` device holds a string of the form

> *m1*/*m2* `memory` *s1*/*s2* `swap`

These give, for each of internal memory and the swapping area, the number of pages used and the total available. These numbers are not blank padded. To turn on swapping, write to `swap` the textual file descriptor number of a file or device on which to swap. See *swap*(8).

The other files served by the *cons* device are all single numbers:

`hz`	frequency of the system clock
`msec`	number of milliseconds since booting
`pgrpid`	process group number
`pid`	process number
`ppid`	parent's process number
`time`	number of seconds since the epoch 00:00:00 GMT, Jan. 1, 1970. (Can be written once, to set at boot time.)

SEE ALSO

bit(3), *keyboard*(6), *auth*(6), *utf*(6)

BUGS

For debugging, two control-T's followed by a letter generate console output: `^T^Tp` prints data about processes, `^T^Tq` prints data about streams, `^T^Tm` prints data about the mount device, `^T^Tb` prints data about the bitblt device, and `^T^Tx` prints date about kernel memory allocation.

The system can be rebooted by typing `^T^Tr.`

NAME

cyc − Cyclone fiber interface

SYNOPSIS

`bind #C /dev`

`/dev/cyc`

DESCRIPTION

The *cyc* device drives the Cyclone CVME961 (*not* 960) card with an attached SQFBR Squall module to provide a high-speed point-to-point 9P link between a CPU server and a file server. Both machines must of course have VME buses. For debugging, the Cyclone may be loaded with on-board software using `xms` (see *con*(1)) and the NINDY ROM supplied with the device. In production, though, it is easiest to replace the ROM with the program in the directory `/sys/src/fs/cyc`. In either case, the on-board software expects an identical Cyclone to be at the other end of the fiber. One of the boards must be in a CPU server, the other in a file server; the systems configure their respective boards dynamically as appropriate.

The driver serves a single file, `/dev/cyc`. When opened, the file initializes the connection to the file server. The resulting file descriptor should be used only to send and receive 9P messages. Typically *boot*(8) will open `/dev/cyc`, prime the connection by sending `nop` and `session` messages (see *attach*(5)), and then `mount` (see *bind*(2)) the file descriptor in the CPU server's name space. Thenceforth all activity on `/dev/cyc` will be mediated by the mount driver *mnt*(3)).

FILES

`/sys/src/fs/cyc`
Directory of on-board software for the Cyclone.

SEE ALSO

CVME960, CVM961 Single Board Computer User's Manual and *SQFBR User's Manual,* Cyclone Microsystems, Inc., New Haven, CT, 1-203-7865536

BUGS

The driver is specific to the SGI Power Series, although the device should operate on any VME bus.

NAME

incon, hsvme, hs386 − Datakit interface

SYNOPSIS

```
bind -a #i /dev
bind -a #h /dev

#i/data
#i/ctl
#h/data
#h/ctl
```

DESCRIPTION

The Datakit interface is a stream directory containing a data and a control file. Each write to the data file is a structured message. The first two bytes of the message are a 9-bit virtual circuit number, low order byte first. The third byte is a control byte. The rest are data bytes. The data bytes are sent onto the Datakit virtual circuit, tagged as data, followed by the control byte, tagged as control.

Messages coming from Datakit are read from the data file in the same format. A read terminates at the end of a message. The largest possible received message is 1024 + 3 bytes.

NAME

dk − Datakit conversations

SYNOPSIS

```
bind #kname /net/dk
bind #iname /net/dk

ctlfd = open(".../ctl", ORDWR);
write(ctlfd, "push dkmux", 10);
write(ctlfd, "config csc [no]restart name nvc window", n);
```

DESCRIPTION

A Datakit device—either `k` for the regular Datakit or `i` for the Incon—is a directory containing up to 256 directories, one per virtual circuit, named `0` through `255,` and a special file named `clone`. The specifier *name* matches the Datakit device to a physical device that its virtual circuits are multiplexed over (see *dkmux* below).

Normally, the standard routines *dial*, *hangup*, *listen*, and *announce* (see *dial*(2)) are used to make, listen for, and control calls over any network. The routines expect the following properties of any multiplexed network, not just Datakit.

Opening the *clone* file is a macro for opening the `ctl` file of an unused virtual circuit. Reading any *ctl* file returns the name of the virtual circuit directory. For example, reading `#k/17/ctl` will return the string `17`.

Each virtual circuit directory contains the files:

`ctl`	to control the virtual circuit: establish a connection, hang it up, etc.
`data`	to converse with the remote end (via read and write)
`listen`	to listen for calls (after announcing; see below)
`other`	information about the conversation
`raddr`	the address of the remote end
`ruser`	the id of the user at the remote end (when applicable)

To set up and tear down virtual circuits a process writes textual commands to the `ctl` file:

`connect` *addr*	connect to address *addr*. If the connection fails, the write returns an error.
`hangup`	tear down a connected virtual circuit.
`announce` *name*	announce the readiness to accept calls to *name*.
`accept` *n*	accept the call on virtual circuit *n*.
`reject` *n e*	reject the call on virtual circuit *n* with error code *e*. *e* must be a number from 0 to 7.

Once a virtual circuit is set up, a process can converse with the remote service by reading and writing the `data` file. Write boundaries are preserved.

Accepting calls to *name* requires the following dance:

1) announce *name* on a virtual circuit.
2) open the `listen` file in that virtual circuit's directory. When a call comes in on a virtual circuit for *name,* the open will return with the file descriptor open to the control file of the incoming virtual circuit.
3) accept or reject the call by writing an `accept` or `reject` command to the `ctl` file of the announced virtual circuit.

A *dkmux* module pushed onto a stream makes that stream a multiplexed connection to a Datakit. The subsequent `config` control message configures the multiplexer and matches it to a *dk* device. The parameters to the `config` message are

csc	the line number of the common signalling channel (must be > 0)
nvc	the number of virtual circuits (optional; default chosen by Datakit)
`[no]restart`	the word `restart` or `norestart` (optional; default is `restart`). `Restart` tells the Datakit to forget all previous connections and authentications for this machine.
name	The name used in binding *dk* device.
window	the default URP window size for virtual circuits on this Datakit line (default is 2048).

FILES

```
#k/clone
#k/[0-255]
#k/[0-255]/data
#k/[0-255]/ctl
#k/[0-255]/listen
#k/[0-255]/ruser
#k/[0-255]/raddr
```

SEE ALSO

stream(3), *dkconfig*(8), *datakit*(3)

NAME

dup − dups of open files

SYNOPSIS

```
bind #d /fd

/fd/0
/fd/1
...
```

DESCRIPTION

The *dup* device serves a one-level directory containing files whose names are decimal numbers. A file of name *n* corresponds to open file descriptor *n* in the current process.

An *open*(5) of file *n* results in a file descriptor identical to what would be returned from a *dup* `(`*n*`, -1)` system call. Note that the result is no longer a file in the *dup* device.

The *stat* operation returns information about the device file, not the open file it points to. A stat of `#d/`*n* will contain *n* for the name, 0 for the length, and 0400, 0200, or 0600 for the mode, depending on whether the dup target is open for reading, writing, or both.

SEE ALSO

dup(2)

NAME

env − environment variables

SYNOPSIS

`bind #e /env`

`/env/`*var1*
`/env/`*var2*
...

DESCRIPTION

The *env* device serves a one-level file directory containing files with arbitrary names and contents. The intention is that the file name is the name of an *environment variable* (see *rc*(1)), and the content is the variable's current value.

When a *fork*(2) system call creates a new process, both the parent and the child continue to see exactly the same files in the *env* device: changes made in either process can be noticed by the other. In contrast, an `rfork` system call with the `RFENVG` bit set (see *fork*(2)) causes a split: initially both process groups see the same environment files, but any changes made in one process group cannot be noticed by the other.

SEE ALSO

rc(1), *fork*(2)

BUGS

A write starting at an offset after the current extent of a file yields an error instead of zero filling.

NAME

fcall − recreate packet delimiters

SYNOPSIS

```
Fctlfd = open(".../ctl", ORDWR);
Fwrite(ctlfd, "push fcall", 10);
```

DESCRIPTION

`Fcall is a` *stream module* (see *stream*(3)) that can be pushed onto a connection to a 9P file server. The function of the module is to recreate packet delimiters lost in transmission. The 9P protocol demands that network connections preserve delimiters between messages written to the file server. Stream based protocols, like TCP, are unable to preserve delimiters. The delimiters must be recreated by the receiver before a packet is read by a file system.

Fcall examines a data stream and identifies 9P messages from their type. The length of the message is computed from the header. Data is collected and buffered by the stream module until an entire 9P message has been assembled. A single message is then delimited and sent upstream to be read by a file server.

SEE ALSO

stream(3), *ip*(3), *exportfs*(4), *srv*(4)

NAME

floppy − floppy disk interface

SYNOPSIS

```
bind #f /dev

/dev/fd0disk
/dev/fd1disk
/dev/fd2disk
/dev/fd3disk
```

DESCRIPTION

The floppy disk interface serves a one-level directory giving access to up to four floppy disk drives. Each drive is represented as a single file. There are no partitions.

NAME

hard, wren − hard disk interface

SYNOPSIS

`bind`
`#w[`*target*`[.`*lun*`]]/dev`

`/dev/hd0disk`
`/dev/hd0partition`
...

DESCRIPTION

The hard disk interfaces (*wren* is a SCSI disk; *hard* is a Safari's internal ST506 disk) serve a one-level directory giving access to the hard disk partitions. The parameter to `attach` defines the numerical SCSI *target* and logical unit number to access. Both default to zero.

Each partition name is prefixed by `hd` and the numeric drive identifier. The partition `disk` always exists and covers the entire disk. The size of each partition as reported by *stat*(2) is the number of bytes in the partition, so the size of `disk` is the size of the entire disk.

The partition `partition` also always exists; it is the last block on the disk. If it contains valid partition data, those partitions will be visible as well. Every time the device is bound, the partitions are updated to reflect any changes in the partition file.

The format of the `partition` file is the string

```
plan9 partitions
```

on a line, followed by a partition specification on a line consisting of a name and textual strings for the block start and limit on the disk.

The program *prep*(8) writes the `partition` table for the disk; its use is preferred to writing it by hand.

SEE ALSO

prep(8), *scsi*(3)

NAME

ip − TCP, UDP, IL network protocols over IP

SYNOPSIS

```
bind -a #Itcp /net
bind -a #Iudp /net
bind -a #Iil /net

#Itcp/tcp/clone
#Itcp/tcp/[0-7]
#Itcp/tcp/[0-7]/data
#Itcp/tcp/[0-7]/ctl
#Itcp/tcp/[0-7]/local
#Itcp/tcp/[0-7]/remote
#Itcp/tcp/[0-7]/status
#Itcp/tcp/[0-7]/listen
...
```

DESCRIPTION

The IP device provides the interface for several protocols that run over IP on an Ethernet. TCP and UDP provide the standard Internet protocols for reliable stream and unreliable datagram communication. IL provides a reliable datagram service for communication between Plan 9 machines. IL is the protocol of choice for most Plan 9 services.

Each of the protocols is served by the IP device, which represents each connection by a set of device files. The top level directory of each protocol contains a `clone` file and subdirectories numbered from zero to the number of connections configured for this protocol.

Opening the `clone` file reserves a connection. The file descriptor returned from the *open*(2) will point to the control file, `ctl`, of the newly allocated connection. Reading the `ctl` file returns a text string representing the number of the connection. Connections may be used either to listen for incoming calls or to initiate calls to other machines.

A connection is controlled by writing text strings to the associated `ctl` file. After a connection has been established data may be read from and written to the data file. For the datagram services, IL and UDP, a `read` of less than the length of a datagram will cause the entire datagram to be consumed. Each write to the data file will send a single datagram on the network. The TCP protocol provides a stream connection that does not preserve `read`/`write` boundaries.

Prior to sending data remote and local addresses must be set for the connection. For outgoing calls the local port number will be allocated randomly if none is set. Addresses are set by writing control messages to the `ctl` file of the connection. The connection is not established until the data file is opened. For IL and TCP the process will block until the remote host has acknowledged the connection. UDP opens always succeed.

The following control messages are supported:

`connect` *ipaddress* ! *port* `[!r]`

Set the remote IP address and port number for the connection. If the *r* flag is supplied and no local address has been specified the system will allocate a restricted port number (less than 1024) for the connection to allow communication with Unix machines login/exec services.

`disconnect`

(UDP only.) Clear the remote address of a UDP connection.

`announce` *port*

Set the local address to *port.* The local IP address can not be set.

`backlog` *n*

(IL and TCP only.) Set the maximum number of pending requests for a given service to *n.* By default *n* is set to five. If more than *n* connections are pending further requests for a service will be rejected.

Port numbers must be in the range 1 to 32767. If a local port has not been announced prior to a `connect` a local port number will be allocated automatically. Local ports are allocated from 5000 up.

Several files report the status of a connection. The `remote` and `local` files contain the IP address and port number for the remote and local side of the connection. The `status` file contains protocol-dependent information to help debug network connections.

A process may accept incoming connections by calling `open` on the `listen` file. The `open` will block until a new connection request arrives. Then `open` will return an open file descriptor which points to the control file of the newly accepted connection. This procedure will accept all calls for the given protocol.

SEE ALSO

listen(8), *dial*(2), *ndb*(6)

NAME

iproute − Internet route table manager

SYNOPSIS

`bind −a #P /net`

`/net/iproute`

DESCRIPTION

The *iproute* device allows the specification of routes for families of IP addresses. It maintains a kernel-resident routing table for IP addresses used by TCP, IL and UDP. Each route consists of a destination IP address, an IP mask, and an IP gateway address. Every packet sent by the system is routed according to the route table. An address matches the route table entry when a packet's destination address matches the table destination address under the mask. When a match is found, the packet is sent to the gateway IP address. If there is no match, the packet is sent with the original destination address. If there are several matches, the one whose mask has the fewest leading zeroes is chosen. (Because of the definition of IP masks, this mask preserves the largest portion of the address and is therefore the most specific.) This is forced by storing the routes in decreasing number of ones order and returning the first match. The default gateway has no ones in the mask and is thus the last matched.

Reading `iproute` reports the current routes entered in the table. Writing control messages to `iproute` edits the table. Route entries are made by writing a string of format

> `add` *ipdest mask ipgateway*

Entries are deleted by writing a string of format

> `delete` *ipdest mask*

The whole table can be cleared by writing the string `flush`.

For example, to install a gateway address to accept all IP packets from a machine:

```
g% echo 'add 0.0.0.0 0.0.0.0 192.20.225.225' > /net/iproute
g% cat /net/iproute
0.0.0.0 & 0.0.0.0 -> 192.20.225.225
```

SEE ALSO

ip(3), *ipconfig*(8)

NAME

kprof − kernel profiling

SYNOPSIS

```
bind −a #T /dev

/dev/kpctl
/dev/kpdata
```

DESCRIPTION

The *kprof* device provides simple profiling data for the operating system kernel. The data accumulates by recording the program counter of the kernel at each 'tick' of the system clock.

The file `kpdata` holds the accumulated counts as 4-byte integers in big-endian byte order. The size of the file depends on the size of kernel text. The first count holds the total number of clock ticks during profiling; the second the number of ticks that occurred while the kernel was running. The rest each hold the number of ticks the kernel program counter was within the corresponding 8-byte range of kernel text, starting from the base of kernel text.

The file `kpctl` controls profiling. Writing the string `"start"` to `kpctl` begins profiling; `"stop"` terminates it. The message `"startclr"` restarts profiling after zeroing the array of counts.

The program *kprof* (see *prof*(1)) formats the data for presentation.

EXAMPLE

The following *rc*(1) script runs a test program while profiling the kernel and reports the results.

```
bind −a '#T' /dev
echo start > /dev/kpctl
runtest
echo stop > /dev/kpctl
kprof /mips/9power /dev/kpdata
```

SEE ALSO

prof(1)

NAME

lance − LANCE Ethernet device

SYNOPSIS

```
bind -a #l /net

#l/ether/clone
#l/ether/[0-7]
#l/ether/[0-7]/data
#l/ether/[0-7]/ctl
#l/ether/[0-7]/stats
#l/ether/[0-7]/type
```

DESCRIPTION

The LANCE Ethernet interface is a directory containing 9 stream directories: one for each of 9 Ethernet packet types and a `clone` file.

Each stream directory contains files to control the stream, receive and send data, and supply statistics. Incoming Ethernet packets are demultiplexed by packet type and passed up the corresponding open stream. Reading from the `data` file reads packets at the head of the stream. A read will terminate at packet boundaries. Each write to the `data` file causes a packet to be sent. The Ethernet address of the interface is inserted into the packet header as the source address.

A stream is assigned a packet type by opening its `ctl` file and writing `connect` *n* where *n* is a decimal integer constant identifying the Ethernet packet type. A value of -1 stands for all types. If multiple streams are assigned to a given packet type a copy of the packet is passed up each stream.

Reading the `type` file returns the decimal value of the assigned Ethernet packet type. Reading the stats file returns status information and the Ethernet address of the interface.

An interface normally receives only packets whose destination address is that of the interface or is the broadcast address, `ff:ff:ff:ff:ff:ff.` The interface can be made to receive all packets on the network by writing the string `promiscuous` to the `ctl` file. The interface remains promiscuous until the control file is closed. The extra packets are passed up only streams of type −1.

NAME

mnt − attach to 9P servers

SYNOPSIS

`#M`

DESCRIPTION

The *mount driver* is used by the `mount` system call (but not `bind`; see *bind*(2)) to connect the name space of a process to the service provided by a 9P server over a communications channel. After the `mount`, system calls involving files in that portion of the name space will be converted by the mount driver into the appropriate 9P messages to the server.

The *mount* system call issues an *auth*(5) message to the server to validate the user and an *attach*(5) message to identify the user of the connection. Each distinct user of a connection must mount it separately; the mount driver multiplexes the access of the various users and their processes to the service.

File-oriented system calls are converted by the kernel into messages in the 9P protocol. Within the kernel, 9P is implemented by procedure calls to the various kernel device drivers. The mount driver translates these procedure calls into remote procedure calls to be transmitted as messages over the communication channel to the server. Each message is implemented by a write of the corresponding protocol message to the server channel followed by a read on the server channel to get the reply. Errors in the reply message are turned into system call error returns.

A *read*(2) or *write* system call on a file implemented by the mount driver may be translated into more than one message, since there is a maximum data size for a 9P message. The system call will return when the specified number of bytes have been transferred or a short reply is returned.

The string `#M` is an illegal file name, so this device can only be accessed directly by the kernel.

BUGS

It is not possible to mount a service through the mount driver across a network. As a result the *window* command will not work from the CPU server since it cannot mount the srv entry for *8½*.

SEE ALSO

bind(2)

NAME

mux − server registry and service multiplexor

SYNOPSIS

`bind #s /srv`

`#s/`*service1*
`#s/`*service2*
...

DESCRIPTION

Mux is a replacement for *srv*(3) that allows a single file server to provide service to processes on both local and remote machines. *Mux* performs all the functions of `srv`.

Plain files created in the top level directory of *mux* behave exactly as described in *srv*(4).

Creating a directory in *mux* produces a stream multiplexer. Many clients may write messages to a single server. *Mux* prefixes each message with a connection number to allow the server to distinguish between clients. Messages written back to *mux* by the server are prefixed by a destination connection number. *Mux* removes the destination connection number before passing messages back to its clients.

When created, a *mux* directory contains two files, `head` and `clone`. Opening the clone file allocates a new connection on the multiplexer. The file descriptor returned is suitable for mounting (see `mount` in *bind*(1) or *bind*(2)). A file named by the new connection number is produced by opening the `clone` file. The numbered connection file may be used to read and write messages to the server. The `head` file should be used by the server to send and receive message from the clients.

A file server must be linked with `/$objtype/lib/libmux.a` to use the device. The library uses the connection numbers provided by the driver to map the fid space of the various client mount drivers into a single fid space for the server. `Libmux` replaces `convS2M` and `convM2S` (see *fcall*(2)) from the C library. A server linked with `libmux` will work correctly with a normal `srv` entry.

Mux's only use is to allow CPU servers to act as a gateway to a file system. It is not normally configured in a system.

BUGS

This should be unnecessary but is required to overcome a failure of vision.

SEE ALSO

bind(2), *srv*(4)

NAME

pipe − two-way interprocess communication

SYNOPSIS

```
bind #| x

x/data
x/ctl
x/data1
x/ctl1
```

DESCRIPTION

An *attach*(5) of this device allocates two new streams joined at the device end. `X/data` and `x/ctl` are the data and control channels of one stream and `x/data1` and `x/ctl1` are the data and control channels of the other stream.

Data written to one channel becomes available for reading at the other. Write boundaries are preserved: each read terminates when the read buffer is full or after reading the last byte of a write, whichever comes first.

Written data is buffered in kernel stream blocks. The writer will block once the stream is full, typically after 32768 bytes or 16 writes. The writer will resume once the stream is less than half full.

If there are multiple writers, each *write* is guaranteed to be available in a contiguous piece at the other end of the pipe. If there are multiple readers, each read will return data from only one write.

The *pipe*(2) system call performs an *attach* of this device and returns file descriptors to the new pipe's `data` and `data1` files. The files are open with mode `ORDWR.`

SEE ALSO

pipe(2)

NAME

proc − running processes

SYNOPSIS

```
bind #p /proc

/proc/n/ctl
/proc/n/mem
/proc/n/note
/proc/n/notepg
/proc/n/proc
/proc/n/segment
/proc/n/status
/proc/n/text
...
```

DESCRIPTION

The *proc* device serves a two-level directory structure. The first level contains numbered directories corresponding to pids of live processes; each such directory contains a set of files representing the corresponding process.

The `mem` file contains the current memory image of the process. A read or write at offset *o,* which must be a valid virtual address, accesses bytes from address *o* up to the end of the memory segment containing *o.* Kernel virtual memory, including the kernel stack for the process and saved user registers (whose addresses are machine-dependent), can be accessed through `mem`. Writes are permitted only while the process is in the `Stopped` state and only to user addresses or registers.

The read-only `proc` file contains the kernel per-process structure. Its main use is to recover the kernel stack and program counter for kernel debugging.

The read-only `status` file contains a string with eight fields, each followed by a space. The fields are: the process name and user name, each 27 characters left justified; the process state, 11 characters left justified; the six 11-character numbers also held in the process's `#c/cputime` file, and the amount of memory used by the process, except its stack, in units of 1024 bytes.

The `text` file is a pseudonym for the file from which the process was executed; its main use is to recover the symbol table of the process.

Textual messages written to the `ctl` file control the execution of the process. Some presume that the process is in a particular state and return an error if it is not.

`stop` Suspend execution of the process, putting it in the `Stopped` state.

`start` Resume execution of a `Stopped` process.

`waitstop`
Do not affect the process directly but, like all other messages ending with `stop`, block the process writing the `ctl` file until the target process is in the `Stopped` state or exits. Also like other `stop` control messages, if the target process would receive a note while the message is pending, it is instead stopped and the debugging process is resumed.

`startstop`
Allow a `Stopped` process to resume, and then do a `waitstop` action.

`hang` Set a bit in the process so that, when it completes an *exec*(2) system call, it will enter the `Stopped` state before returning to user mode. This bit is inherited across a *fork*(2).

`kill` Kill the process with extreme prejudice.

Strings written to the `note` file will be posted as a note to the process (see *notify*(2)). The note should be less than `ERRLEN−1` characters long; the last character is reserved for a terminating NUL character. A read of at least `ERRLEN` characters will retrieve the oldest note posted to the process and prevent its delivery to the process. The `notepg` file is similar, but the note will be delivered to all the processes in the target process's *note group* (see *fork*(2)). The `notepg` file is write-only.

FILES

`/sys/src/9/*/mem.h`
`/sys/src/9/*/dat.h`

SEE ALSO

cons(3)

NAME

root − the root file system

SYNOPSIS

`/`
`/boot`
`/dev`
`/env`
`/proc`
`/net`

DESCRIPTION

The syntax `#/` is illegal, so this device can only be accessed directly by the kernel.

This device is set up by the kernel to be the root of the name space. The names in the one-level tree are mostly just place-holders, to allow a place to *bind*(2) to. The exception is `/boot`, which provides executable code when read. The kernel does an *exec*(2) of `/boot` when initializing.

NAME

rtc − real-time clock and non-volatile RAM

SYNOPSIS

```
bind #r /dev
```

```
/dev/rtc
/dev/nvram
```

DESCRIPTION

The *rtc* device supports the Mostek MK48T12-15 Zeropower/Timekeeper and similar devices with real-time clocks and non-volatile RAM.

The `rtc` file behaves just like `/dev/time` (see *cons*(3)). The real-time clock is maintained on-board; `/dev/time` is set from the file server. Neither is necessarily more accurate.

The `nvram` file provides (if permission allows) access to the local non-volatile RAM. For example, *boot*(8) reads the machine's key (see *auth*(8)) from there.

SEE ALSO

auth(8), *boot*(8)

NAME

scc, duart, uart − serial communication control

SYNOPSIS

```
bind −a #t /dev

/dev/eia0
/dev/eia0ctl
/dev/eia1
/dev/eia1ctl
```

DESCRIPTION

The serial line devices serve a one-level directory, giving access to the serial ports. There are several devices serving the same files; the particular one used depends on the machine involved. `Eia0` is a stream data file. It can be read and written to use that port. Reads will block until at least one character is available. `Eia0ctl` is a stream control file associated with the port. `Eia1` and `eia1ctl` are similar, but for a second serial line.

The `ctl` file can be used to push stream modules onto the port. One can also write one of the following textual commands to a `ctl` file:

`b`*n*	set the baud rate to *n*.
`d`*n*	set dtr if *n* is non-zero; else clear it.
`k`*n*	send a break lasting *n* milliseconds.
`r`*n*	set RTS if *n* is non-zero; else clear it.
`m`*n*	obey modem CTS signal *n* is non-zero; else clear it.
`p`*c*	set parity to odd if *c* is `o`, to even if *c* is `e`; else set no parity.
`s`*n*	set number of stop bits to *n*. Legal values are 1 or 2.
`l`*n*	set number of bits per byte to *n*. Legal values are 5, 6, 7, or 8.

NAME

scsi − SCSI command interface

SYNOPSIS

`#S/id`
`#S/0/cmd`
`#S/0/data`
`#S/0/debug`
...

DESCRIPTION

The *scsi* interface is accessed through a two-level directory. The single-byte `id` file contains the SCSI id of the host interface, typically 7. Some implementations allow this to be changed by writing to the file; in many cases, the higher-order bits are hardware specific.

Each SCSI target *n* (0≤*n*≤7) is associated with a subdirectory `#S/`*n* containing files `cmd`, `data`, and `debug`. The following steps may be used to execute a SCSI command:

> The command block is written to the `cmd` file.
>
> The `data` file is either written or read depending on the direction of the transfer. (A command that involves no data transfer is executed with a zero-length write.)
>
> The `cmd` file is read to retrieve the status of the command, returned as a 4-byte big-endian integer.

Writing an ASCII `1` to the `debug` file causes tracing information to be written to `/dev/klog`; writing a `0` turns the tracing off.

SEE ALSO

hard(3)

NAME

srv − server registry

SYNOPSIS

`bind #s /srv`

`#s/`*service1*
`#s/`*service2*
...

DESCRIPTION

The *srv* device provides a one-level directory holding already-open channels to services. In effect, *srv* is a bulletin board on which processes may post open file descriptors to make them available to other processes.

To install a channel, `create` a new file such as `/srv/myserv` and then write a text string (suitable for *strtoul*; see *atof*(2)) giving the file descriptor number of an open file. Any process may then open `/srv/myserv` to acquire another reference to the open file that was registered.

An entry in *srv* holds a reference to the associated file even if no process has the file open. Removing the file from `/srv` releases that reference.

It is an error to write more than one number into a server file, or to create a file with a name that is already being used.

EXAMPLE

To drop one end of a pipe into `/srv`, that is, to create a named pipe:

```
int fd, p[2];
char buf[32];

pipe(p);
fd = create("/srv/namedpipe", 1, 0666);
sprint(buf, "%d", p[0]);
write(fd, buf, strlen(buf));
close(fd);
close(p[0]);
write(p[1], "hello", 5);
```

At this point, any process may open and read `/srv/namedpipe` to receive the `hello` string. Data written to `/srv/namedpipe` will be received by executing

```
read(p[1], buf, sizeof buf);
```

in the above process.

NAME

stream − a structure for communications

SYNOPSIS

`x/data`
`x/ctl`

DESCRIPTION

A *stream* is not a device per se. However, many devices use the *streams* package in the kernel to implement communications channels. The properties described here are common to all such channels.

All streams are represented by two standard files, `ctl` and `data,` plus any others the particular device wants to add. Reading and writing the `data` file receives and sends data on the channel. If the channel is message oriented, each write will represent a message and each read will return at most one message. If the buffer given in a read is smaller than the message, subsequent reads will return the remainder of the message.

Writing textual command strings to the `ctl` file performs control operations on the stream. The strings need not be null-terminated. Each device may add to the control operations. The common control operations are:

`hangup`	Hang up this stream. Any subsequent writes will return an error. The first subsequent read will return 0. All following ones will return an error.
`push` *name*	Push the module *name* onto the top of the stream.
`pop`	Pop the top module off the stream

Reading the `ctl` file returns a textual identifier for the stream. This is used by multiplexed devices and its use is described with the particular device.

SEE ALSO

pipe(3), *dk*(3), *cons*(3), *async*(3)

NAME

vga − vgasize, vgatype, vgaport

SYNOPSIS

```
bind #v /dev

/dev/vgasize
/dev/vgatype
/dev/vgaport
```

DESCRIPTION

The vga device allows configuration of a graphics controller on a pc.

Writing a string to the file `vgasize` of the form: *xmax*X*ymax*X*bits*, where *xmax*, *ymax* and *bits* are numbers, tells the kernel the width and height of the screen in pixels and the number of bits per pixel. Reading `vgasize` returns the current setting.

Writing a string to `vgatype` tells the kernel what kind of controller is being used. The possibilities are currently *generic* and *tseng*.

Vgaport provides user level access to the byte port space. Reads and writes to offsets in this file perform 386 inb and outb operations on ports equal to the offset. Offsets below 0x300 are illegal. This device is used by the command *vga*(8) to configure a VGA controller chip.

NAME

intro − introduction to file servers

DESCRIPTION

A Plan 9 *file server* provides a file tree to processes. This section of the manual describes servers than can be mounted in a name space to give a file-like interface to interesting services. A file server may be a provider of a conventional file system, with files maintained on permanent storage, or it may also be a process that synthesizes files in some manner.

SEE ALSO

bind(1)

NAME

8½ − window system files

SYNOPSIS

`8½ [ -i 'cmd' ] [ -s ] [ -f font ]`

The window system *8½* serves a variety of files for reading, writing, and controlling windows. Some of them are virtual versions of system files for dealing with the display, keyboard, and mouse; others control operations of the window system itself. *8½* posts its service in the `/srv` directory, using a name constructed from a catenation of the user ID and a process id; the environment variable `$8½srv` is set to this service name within processes running under the control of each invocation of *8½*.

A *mount* (see *bind*(1)) of that file causes `8½` to create a new window; the attach specifier in the *mount* gives the coordinates of the created window. The syntax of the specifier is

> `N` *pid minx miny maxx maxy*

where *pid* is the process id of a process in the note group (see *fork*(2)) to receive `interrupt` and `hangup` notes in that window.

When a window is created either by the *window* command (see *8½*(1)) or by using the menu supplied by *8½*, this server is mounted on `/mnt/8½` and also `/dev`; the files mentioned here appear in both those directories.

Some of these files supply virtual versions of services available from the underlying environment, in particular the character terminal files *cons*(3), and all the bit devices *bit*(3), each specific to the window. Other files are unique to *8½*.

`bitblt`
: is a virtual version of the `bitblt` file within the current window; see *bit*(3), *graphics*(2). All operations are clipped to the current window. The coordinate system is absolute; it refers to the real screen.

`cons` is a virtual version of the standard terminal file *cons*(3). *8½* supplies extra editing features and a scroll bar (see *8½*(1)).

`consctl`
: controls interpretation of keyboard input. Writing strings on it sets these modes: `rawon` turns on raw mode; `rawoff` turns off raw mode; `holdon` turns on hold mode; `holdoff` turns off hold mode. Closing the file makes the window revert to default state (raw off, hold off).

`label` initially contains a string with the process ID of the lead process in the window and the command being executed there. It may be written and is used as a tag when the window is hidden.

`mouse` is a virtual version of the standard mouse file (see *bit*(3)). Opening it turns off scrolling, editing, and *8½*-supplied menus in the associated window. The 0x80 bit in the buttons byte of a returned record indicates that the window has been reshaped. Reading this file blocks until the mouse moves or a button changes. Mouse movements or button changes are invisible when the mouse cursor is located outside the window.

`nbmouse`
: is a non-blocking version of `mouse`; it always returns the current state. Its use is discouraged.

`select`
: returns the selected text in the designated window. It may not be written.

`snarf` returns the string currently in the snarf buffer. Writing this file sets the contents of the snarf buffer.

`text` returns the full contents of the window. It may not be written.

`winid` returns the unique and unchangeable ID for the window; it is a string of digits.

`window`
: is the virtual version of `/dev/screen`; see *bit*(3). It contains the depth, coordinates, and bitmap corresponding to the associated window.

`windows`
: is a directory containing a subdirectory for each window, named by the unique ID for that window. Within each subdirectory are entries corresponding to several of the special files associated with that window: `bitblt`, `cons`, `consctl`, `label`, `mouse`, `nbmouse`, `select`, `window`.

EXAMPLES

Create a window to be created in the upper left corner, and the word `hi` to be printed there.

```
mount $8½srv /tmp N$pid' 0 0 128 64'
echo hi > /tmp/cons
```

Print the text currently selected in window 123.

```
cat /dev/windows/123/select
```

SEE ALSO

8½(1), *bit*(3), *cons*(3), *event*(2), *graphics*(2).

NAME

cfs − cache file system

SYNOPSIS

`cfs -s` [`-rd`] [`-f` *partition*]

`cfs -a` *netaddr* [`-rd`] [`-f` *partition*] [*mtpt*]

DESCRIPTION

Cfs is a user-level file server that caches information about remote files onto a local disk. It is normally started by the kernel at boot time, though users may start it manually. *Cfs* is interposed between the kernel and a network connection to a remote file server to improve the efficiency of access across slow network connections such as modem lines. On each open of a file *cfs* checks the consistency of cached information and discards any old information for that file.

Cfs mounts onto *mtpt* (default `/`) after connecting to the file server.

The options are:

`s` the connection to the remote file server is on file descriptors 0 and 1.

`a` *netaddr*
dial the destination *netaddr* to connect to a remote file server.

`r` reformat the cache disk partition.

`d` turn on debugging

`f` *partition*
use file *partition* as the cache disk partition.

All 9P messages except `read`, `clone`, and `walk` (see *intro*(5)) are passed through *cfs* unchanged to the remote server. A `clone` followed immediately by a `walk` is converted into a `clwalk`. If possible, a `read` is satisfied by cached data. Otherwise, the file server is queried for any missing data.

FILES

`/dev/hd0cache`
Default file used for storing cached data.

NAME

dossrv, 9660srv, eject − DOS and ISO9660 file systems

SYNOPSIS

`dossrv` [`−v`] [`−s`] [`−f` *file*] [*service*]

`9660srv` [`−v`] [`−s`] [`−f` *file*] [*service*]

`eject` [n]

DESCRIPTION

Dossrv is a file server that interprets DOS file systems. A single instance of *dossrv* can provide access to multiple DOS disks simultaneously.

Dossrv posts a file descriptor named *service* (default `dos`) in the `/srv` directory. To access the DOS file system on a device, use `mount` with the *spec* argument (see *bind*(1)) the name of the file holding raw DOS file system, typically the disk. If *spec* is undefined in the `mount`, *dossrv* will use *file* as the default name for the device holding the DOS system.

Normally *dossrv* creates a pipe to act as the communications channel between itself and its clients. The `−s` flag instructs *dossrv* to use its standard input and output instead. The kernels use this if they are booting from a DOS disk. This flag also prevents the creation of an explicit service file in `/srv`.

The `−v` flag causes verbose output for debugging.

9660srv is identical to *dossrv* in specification, except that it interprets ISO9660 CD-ROM file systems instead of DOS file systems.

Eject ejects a floppy from drive *n*, default 0.

EXAMPLE

Mount a floppy disk with a DOS file system on it.

```
dossrv
mount -c /srv/dos /n/a: /dev/fd0disk
```

SEE ALSO

kfs(4)

NAME

exportfs − network file server plumbing

SYNOPSIS

`exportfs` [`-a`] [`-c` *ctlfile*]

DESCRIPTION

Exportfs is a user level file server that allows Plan 9 compute servers, rather than file servers, to export portions of a name space across networks. The service is started either by the *cpu*(1) command or by a network listener process. An initial protocol establishes a root directory for the exported name space. The connection to *exportfs* is then mounted. *Exportfs* then acts as a relay file server: operations in the imported file tree are executed on the remote server and the results returned. This gives the appearance of exporting a name space from a remote machine into a local file tree.

The `-a` option instructs *exportfs* to authenticate the user, usually because it is being invoked from a remote machine.

The `-c` options specifies a network control file onto which *exportfs* will push the *fcall* line discipline. This option is intended for networks that do not preserve read/write boundaries.

The `cpu` command uses *exportfs* to serve device files in the terminal. The *import*(4) command calls *exportfs* on a remote machine, permitting users to access arbitrary pieces of name space on other systems.

NAME

fs − file server, bootes

SYNOPSIS

none

DESCRIPTION

The file server is the main file system for Plan 9. It is a stand-alone system that runs on a separate computer. It serves the Plan 9 protocol on a variety of networks including Datakit/URP, Ethernet IL/IP and Cyclone fiber direct connections. The name of this machine is `bootes`.

`Bootes` can use an external authentication server to validate clients. Because the authentication server itself uses `bootes` as a file server, though, to avoid chicken-and-egg problems `bootes` usually authenticates its own connections. Thus when changes are made to the authentication server's database it is necessary to run `auth` (see *fs*(8)) to update `bootes`'s internal state.

The user `none` is always allowed to attach to `bootes` without authentication but has minimal permissions.

Bootes maintains three file systems on a combination of disks and write-once-read-many (WORM) magneto-optical disks.

`other` is a simple disk-based file system similar to *kfs*(4).

`main` is a worm-based file system with a disk-based look-aside cache. The disk cache holds modified worm blocks to overcome the write-once property of the worm. The cache also holds recently accessed non-modified blocks to speed up the effective access time of the worm. Occasionally (usually daily at 5AM) the modified blocks in the disk cache are *dumped*. At this time, traffic to the file system is halted and the modified blocks are relabeled to the un-written portion of the worm. After the dump, the file system traffic is continued and the relabeled blocks are copied to the worm by a background process.

`dump` Each time the main file system is dumped, its root is appended to a subdirectory of the dump file system. Since the dump file system is not mirrored with a disk cache, it is read-only. The name of the newly added root is created from the date of the dump: */yyyy/mmdds.* Here *yyyy* is the full year, *mm* is the month number, *dd* is the day number and *s* is a sequence number if more than one dump is done in a day. For the first dump, *s* is null. For the subsequent dumps *s* is 1, 2, 3 etc.

The root of the main file system that is frozen on the first dump of March 1, 1992 will be named `/1992/0301/` in the dump file system.

EXAMPLES

Place the root of the `dump` file system on `/n/dump` and show the modified times of the mips C compiler over all dumps in February, 1992:

```
9fs dump
ls -l /n/dump/1992/02??/mips/bin/vc
```

To get only one line of output for each version of the compiler:

```
ls -lp /n/dump/1992/02??/mips/bin/vc | uniq
```

Make the `other` file system available in directory `/n/bootesother`:

```
mount -ct /srv/boot /n/bootesother other
```

SEE ALSO

yesterday(1), *srv*(4), *fs*(8)
Sean Quinlan, *A Cached WORM File System,* Software − Practice and Experience, December, 1991

NAME

ftpfs, ftp − file transfer protocol (FTP) file system

SYNOPSIS

`ftpfs` [`-/`] [`-m` *mountpoint*] [`-a` *password*] *system*

DESCRIPTION

Ftpfs dials the TCP file transfer protocol (FTP) port, 21, on *system* and mounts itself (see *bind*(2)) on *mountpoint* (default `/n/ftp`) to provide access to files on the remote machine. If required by the remote machine *ftpfs* will prompt for user name and password. The user names `ftp` and `anonymous` are often used to offer guest/read-only access to machines. Anonymous ftp may be called without user interaction by specifying the *password*.

By default the file seen at the mount point is the user's home directory. The option `-/` can be used to force the mount point to correspond to the remote root.

To terminate the connection, `unmount` (see *bind*(1)) the mount point.

SEE ALSO

bind(2)

BUGS

Symbolic links on remote Unix systems will always have mode 0777 and a length of 8.

After connecting to a TOPS-20 system, the mountpoint will contain only one directory, usually `/n/ftp/PS:<ANONYMOUS>`. However, walking to any valid directory on that machine will succeed and cause that directory entry to appear under the mountpoint.

NAME

help − make and control help windows

SYNOPSIS

`help/help`

Help(1) serves a variety of files for reading, writing, and controlling its windows. The files are mounted on the directory `/mnt/help` and are also unioned into the beginning of `/dev`. At the top level of that directory are files `cons`, `consctl`, and `mouse`, much like those of *8½*(1). *Help's* versions of these files have restricted utility, however: `cons` may be written to but not read and both `consctl` and `mouse` draw an error when opened. When a program is run from within *help*, its standard input is connected to `/dev/null` and its standard output to `/mnt/help/cons`. Text written to `cons` will appear in *help's* `Errors` window. `Consctl` and `mouse` are there to prevent access to the files *help* is itself using.

Another file in `/mnt/help` is `index`, which holds one line of text for each *help* window. Each line has the numeric window id for the window, a tab, and the tag of the window.

The rest of the files in `/mnt/help` are directories, one per window, named by the numeric window id of the corresponding window. Each directory contains the following files:

`body`	This read-only file contains the text of the body of the window.
`bodyapp`	Text written to this write-only file is appended to the body.
`bodypos`	This read-only file contains the starting and ending byte offsets of the selection in the body, formatted as two textual numbers in 12-byte blank padded fields.
`bodysel`	This read-only file contains the selected text in the body.
`tag`	This read-only file contains the text of the tag of the window.
`tagpos`	Like `bodypos` but for the tag.
`tagsel`	Like `bodysel` but for the tag.
`ctl`	A read of this file returns the textual numeric id of the window. Messages written to `ctl` change the contents of the window. Each message must be contained in a single 9P write message (see *read*(5)) and must begin with an ASCII letter, possibly an address, and a newline. An address is one or two comma-separated positions in the file, either line numbers or character positions. The address syntax is a subset of that in *sam*(1). If the message contains text to be added to the window, the text follows the newline and need not be newline-terminated. This description mentions only the upper case versions. The symbol `\n` stands for a literal newline:

`a\n`<*text*>	Append the text to the tag.
`A\n`<*text*>	Append the text to the body.
`d`<*address*>`\n`	Delete the addressed text from the tag.
`D`<*address*>`\n`	Delete the addressed text from the body.
`i`<*address*>`\n`<*text*>	Insert the text in the tag at the addressed location. If the address has two components, the second is ignored.
`I`<*address*>`\n`<*text*>	Like *i*, but insert in the body.
`u\n`	Mark the window 'not dirty', that is, remove the `Put!` string from the tag.

A single directory `/mnt/help/new`, when accessed, creates a new window. The easiest way to manage a new window is to open `/mnt/help/new/ctl`, which will create it automatically, and then read the file to retrieve the numeric id. The other files in the window may then be opened as usual.

EXAMPLE

Create a new window and run *date*(1) in it.

```
id = `{cat /mnt/help/new/ctl}
echo 'a
Current time    Close!' > /mnt/help/$id/ctl
date > /mnt/help/$id/bodyapp
```

SEE ALSO

help(1), *8½*(4)

NAME

import − import a name space from a remote system

SYNOPSIS

`import` [*option ...*] *system file* [*mountpoint*]

DESCRIPTION

Import allows an arbitrary *file* on a remote *system* to be imported into the local name space. Usually *file* is a directory, so the complete file tree under the directory is made available.

A process is started on the remote machine, with authority of the user of *import,* to perform work for the local machine using the *exportfs*(4) service. If *mountpoint* is omitted *import* uses the name of the remote *file* as the local mount point.

If *file* is a directory, *import* allows options exactly as in *mount* and *bind*(1)) to control the construction of union directories.

EXAMPLE

To allow a Datakit-only machine to access an Ethernet using TCP:

```
import -a kremvax /net
con tcp!ucbvax
```

or

```
import -a kremvax /net
echo 'add tcp' > /net/cs
con ucbvax
```

SEE ALSO

bind(1), `cs` in *ndb*(8)

NAME

iostats − file system to measure I/O

SYNOPSIS

`iostats` *cmd* [*args...*]

DESCRIPTION

Iostats is a user-level file server that interposes itself between a program and the regular file server, which allows it to gather statistics of file system use at the level of the Plan 9 file system protocol, 9P. After a program exits a report is printed on standard error.

The report consists of three sections. The first section reports the amount of user data in `read` and `write` messages sent by the program and the average rate at which the data was transferred. The `protocol` line reports the amount of data sent as message headers, that is, protocol overhead. The `rpc` line reports the total number of file system transactions.

The second section gives the number of messages, the fastest, slowest, and average turn around time and the amount of data involved with each 9P message type. The final section gives an I/O summary for each file used by the program in terms of opens, reads and writes.

BUGS

Poor clock resolution means that large amounts of I/O must be done to get accurate rate figures.

NAME

keyfs − authentication database files

SYNOPSIS

`keyfs` [`-d`] [`-m`*mntpt*] [`-k`*key*] [*keyfile*]

DESCRIPTION

Keyfs serves a two-level file tree for manipulating authentication information. It runs on the machine providing authentication service for the local Plan 9 network, which may be a dedicated authentication server or a CPU server. The programs described in *auth*(8) use *keyfs* as their interface to the authentication database.

Keyfs reads and decrypts file *keyfile* (default `/adm/keys`) using the DES key *key*, which is by default read from `#r/nvram` (see *rtc*(3)). With option `-d`, keyfs uses `/dev/crypt` for decryption. *Keyfile* holds a 41-byte record for each user in the database. Each record is encrypted separately and contains the user's name, DES key, status, host status, and expiration date. The name is a null-terminated UTF string `NAMELEN` bytes long. The status is a byte containing binary 0 if the account is enabled, 1 if it is disabled. Host status is a byte containing binary 1 if the user is a host, and 0 otherwise. The expiration date is four-byte little-endian integer which represents the time in seconds since the epoch (see *date*(1)) at which the account will expire. If any changes are made to the database that affect the information stored in *keyfile*, a new version of the file is written.

There are two authentication databases, one for Plan 9 user information, and one for SecureNet user information. A user need not be installed in both databases but must be installed in the Plan 9 database to connect to a Plan 9 server.

Keyfs serves an interpretation of the *keyfile* in the file tree rooted at *mntpt* (default `/mnt/keys`). Each user *user* in *keyfile* is represented as the directory *mntpt*/*user*.

Making a new directory in *mntpt* creates a new user entry in the database. Removing a directory removes the user entry, and renaming it changes the name in the entry. *Keyfs* does not allow duplicate names when creating or renaming user entries.

All files in the user directories except for `key` contain UTF strings with a trailing newline when read, and should be written as UTF strings with or without a trailing newline. `Key` contains the `DESKEYLEN`-byte encryption key for the user.

The following files appear in the user directories.

`key` The authentication key for the user. If the user's account is disabled or expired, reading this file returns an error. Writing *key* changes the key in the database.

`log` The number of consecutive failed authentication attempts for the user. Writing the string `bad` increments this number; writing `good` resets it to 0. If the number reaches fifty, *keyfs* disables the account. Once the account is disabled, the only way to enable it is to write the string `ok` to `status`. This number is not stored in *keyfile*, and is initialized to 0 when *keyfs* starts.

`status`
The current status of the account, either `ok` or `disabled`. Writing `ok` enables the account; writing `disabled` disables it.

`expire`
The expiration time for the account. When read, it contains either the string `never` or the time in seconds since the epoch that the account will expire. When written with strings of the same form, it sets the expiration date for the user. If the expiration date is reached, the account is not disabled, but *key* cannot be read without an error.

`ishost`
This file exists only if the user is a host (the host status for the user is 1). Hosts are the only users able to receive calls. Creating it makes the user a host and sets the host status to 1, and removing it sets the host status to 0.

FILES

`/adm/keys`
Encrypted key file for the Plan 9 database.

`/adm/netkeys`
Encrypted key file for the SecureNet database.

`#r/nvram`
The non-volatile RAM on the server, which holds the key used to decrypt key files.

SEE ALSO

auth(6), *namespace*(6), *auth*(8)

NAME

kfs − disk file system

SYNOPSIS

`disk/kfs` [`−rc`] [`−b` *n*] [`−f` *file*] [`−n` *name*] [`−s`]

DESCRIPTION

Kfs is a local user-level file server for a Plan 9 terminal with a disk. `Kfs` begins by checking the file system for consistency, rebuilding the free list, and placing a file descriptor in /srv/*service*, where *service* is the service name (default *kfs*). If the file system is inconsistent, the user is asked for permission to ream (*q.v.*) the disk. The file system is not checked if it is reamed.

The options are

`b` *n*	If the file system is reamed, use *n* byte blocks. Larger blocks make the file system faster and less space efficient. `1024` and `4096` are good choices. *N* must be a multiple of 512.
`c`	Do not check the file system.
`f` *file*	Use *file* as the block storage file. The default is `/dev/hd0fs`.
`n` *name*	Use kfs.*name* as the name of the service.
`r`	Ream the file system, erasing all of the old data and adding all blocks to the free list.
`s`	Post file descriptor zero in `/srv/`*service* and read and write protocol messages on file descriptor one.

EXAMPLES

```
% kfs -rb4096 -nlocal
% mount -c /srv/kfs.local /n/kfs
```

Create a file system with service name *kfs.local* and mount it on `/n/kfs`.

FILES

`/dev/hd0fs`
Default file holding blocks.

SEE ALSO

kfscmd(8), *mkfs*(8), *prep*(8), *hard*(3)

BUGS

This file system is known to be unreliable and shouldn't be depended on.

NAME

namespace − structure of conventional file name space

SYNOPSIS

none

DESCRIPTION

After a user's profile has run, the file name space should adhere to a number of conventions if the system is to behave normally. This manual page documents those conventions by traversing the file hierarchy and describing the points of interest. It also serves as a guide to where things reside in the file system proper. The traversal is far from exhaustive.

First, here is the appearance of the file server as it appears before any mounts or bindings.

`/`	The root directory.
`/adm`	The administration directory for the file server.
`/adm/users`	List of users known to the file server; see *users*(6).
`/adm/keys`	Authentication keys for users.
`/adm/netkeys`	SecureNet keys for users; see *securenet*(8).
`/adm/timezone`	Directory of timezone files; see *ctime*(2).
`/adm/timezone/EST.EDT`	Time zone description for Eastern Time. Other such files are in this directory too.
`/adm/timezone/timezone`	Time zone description for the local time zone; a copy of one of the other files in this directory.
`/bin`	
`/dev`	
`/env`	
`/fd`	
`/net`	
`/proc`	
`/srv`	
`/tmp`	All empty unwritable directories, place holders for mounted services and directories.
`/mnt`	A directory containing mount points for applications.
`/n`	A directory containing mount points for file trees imported from remote systems.
`/68020`	
`/386`	
`/sparc`	
`/960`	
`/hobbit`	
`/mips`	Each CPU architecture supported by Plan 9 has a directory in the root containing architecture-specific files, to be selected according to `$objtype` or `$cputype` (see *2c*(1) and *init*(8)). Here we list only those for `/mips`.
`/mips/init`	The initialization program used during bootstrapping; see *init*(8).
`/mips/bin`	Directory containing binaries for the MIPS architecture.
`/mips/bin/aux`	
`/mips/bin/games`	
etc.	Subdirectories of `/mips/bin` containing auxiliary tools and collecting related programs.
`/mips/lib`	Directory of object code libraries as used by `vl` (see *2l*(1)).
`/mips/include`	Directory of MIPS-specific C include files.
`/mips/9*`	The files in `/mips` beginning with a `9` are binaries of the operating system.
`/mips/mkfile`	Selected by *mk*(1) when `$objtype` is `mips`, this file configures `mk` to compile for the MIPS architecture.
`/rc`	Isomorphic to the architecture-dependent directories, this holds executables and libraries for the shell, *rc*(1).

`/rc/bin`	Directory of shell executable files.
`/rc/lib`	Directory of shell libraries. Holds only one file, `rcmain`.
`/lib`	Collections of data, generally not parts of programs.
`/lib/bible`	The King James edition
`/lib/chess`	
`/lib/sky`	
etc.	Databases.
`/lib/ndb`	The network database used by the networking software; see *ndb*(6) and *ndb*(8).
`/lib/namespace`	The file used by `newns` (see *auth*(2)) to establish the default name space; see *namespace*(6).
`/sys`	System software.
`/sys/lib`	Pieces of programs not easily held in the various `bins`.
`/sys/lib/troff`	Directory of *troff*(1) font tables and macros.
`/sys/lib/yaccpar`	The *yacc*(1) parser.
`/sys/man`	The manual.
`/sys/doc`	Other system documentation.
`/sys/log`	Log files created by various system services.
`/sys/src`	Top-level directory of system sources.
`/sys/src/cmd`	Source to the commands in the `bin` directories.
`/sys/src/9`	Source to the operating system for terminals and CPU servers.
`/sys/src/fs`	Source to the operating system for file servers.
`/sys/src/lib*`	Source to the libraries.
`/mail`	Directory of electronic mail; see *mail*(1).
`/mail/box`	Directory of users' mail box files.
`/mail/lib`	Directory of alias files, etc.
`/mail/log`	Directory of mail log files.
`/mail/log/status`	Log of mail activity.

The following files and directories are modified in the standard name space, as defined by `/lib/namespace` (see *namespace*(6)).

`/`	The root of the name space. It is a kernel device, *root*(3), serving a number of local mount points such as `/bin` and `/dev` as well as the bootstrap program `/boot`. Unioned with `/` is the root of the main file server.
`/boot`	Compiled into the operating system kernel, this file establishes the connection to the main file server and starts `init`; see *boot*(8) and *init*(8).
`/bin`	Mounted here is a union directory composed of `/$objtype/bin`, `/rc/bin`, `$home/$objtype/bin`, etc., so `/bin` is always the directory containing the appropriate executables for the current architecture.
`/dev`	Mounted here is a union directory containing I/O devices such as the console (*cons*(3)), the bitmap display (*bit*(3)), etc. The window system, *8½*(1), prefixes this directory with its ownn version, overriding many device files with its own, multiplexed simulations of them.
`/env`	Mounted here is the environment device, *env*(3), which holds environment variables such as `$cputype`.
`/net`	Mounted here is a union directory formed of all the network devices available.
`/net/cs`	The communications point for the connection server, `ndb/cs` (see *ndb*(8)).
`/net/il`	
`/net/tcp`	
`/net/udp`	Directories holding the IP protocol devices (see *ip*(3)).

`/net/dk`	A directory holding the Datakit protocol devices (see *dk*(3)).
`/proc`	Mounted here is the process device, *proc*(3), which provides debugging access to active processes.
`/fd`	Mounted here is the dup device, *dup*(3), which holds pseudonyms for open file descriptors.
`/srv`	Mounted here is the service registry, *srv*(3), which holds connections to file servers.
`/srv/boot`	The communication channel to the main file server for the machine.
`/mnt/8½`	Mount point for the window system.
`/mnt/term`	Mount point for the terminal's name space as seen by the CPU server after a *cpu*(1) command.
`/n/kremvax`	A place where machine `kremvax`'s name space may be mounted.
`/tmp`	Mounted here is each user's private `tmp,` `$home/tmp`.

SEE ALSO

intro(1), *namespace*(6)

NAME

ramfs − memory file system

SYNOPSIS

`ramfs` [`-i`] [`-s`] [*mountpoint*]

DESCRIPTION

Ramfs starts a process that mounts itself (see *bind*(2)) on *mountpoint* (default `/tmp`). The *ramfs* process implements a file tree rooted at *dir*, keeping all files in memory. Initially the file tree is empty.

The `-i` flag tells *ramfs* to use file descriptors 0 and 1 for its communication channel rather than create a pipe. This makes it possible to use *ramfs* as a file server on a remote machine: the file descriptors 0 and 1 will be the network channel from *ramfs* to the client machine. The `-s` flag causes *ramfs* to post its channel on `/srv/ramfs` rather than mounting it on *mountpoint*, enabling multiple clients to access its files. However, it does not authenticate its clients and its implementation of groups is simplistic, so it should not be used for precious data.

This program is useful mainly as an example of how to write a user-level file server. It can also be used to provide high-performance temporary files.

SEE ALSO

bind(2)

NAME

srv, 9fs, dk232, dkmodem − start network file service

SYNOPSIS

`srv` [`-m`] [`-t`] [*net*!]*system*[!*service* [*srvname* [*mtpt*]]

`9fs -t` [*net*!]*system* [*mountpoint*]

`dk232` [*server*]

`dkmodem` [*telno*]

DESCRIPTION

Srv dials the given machine and initializes the connection to serve the 9P protocol. It then creates in `/srv` a file named *srvname*. Users can then `mount` (see *bind*(1)) the service, typically on a name in `/n`, to access the files provided by the remote machine. If *srvname* is omitted, the first argument to `srv` is used. Option `m` directs *srv* to mount the service on `/n/`*system* or onto *mtpt* if it is given. If option `t` is given, the mount is authenticated.

The specified *service* must serve 9P. Usually *service* can be omitted; when calling some non-Plan 9 systems, a *service* such as `u9fs` must be mentioned explicitly.

The *9fs* command does the *srv* and the *mount* necessary to make available the files of *system* on network *net.* The files are mounted on *mountpoint*, if given; otherwise they are mounted on `/n/`*system.* If *system* contains `/` characters, only the last element of *system* is used in the `/n` name. With option `t` the mount is authenticated.

Dk232 configures a serial line as a Datakit device and connects to a file server (default `bootes`) using `9fs`.

Dkmodem dials a file server at telephone number `telno` and configures the line as a Datakit device using `9fs`.

EXAMPLES

To see kremvax's and deepthought's files in `/n/kremvax` and `/n/deepthought`:

```
9fs kremvax
9fs hhgttg/deepthought
```

NOTE

The TCP port used for 9P is 564.

SEE ALSO

bind(1), *dial*(2), *srv*(3), *ftpfs*(4), *dkconfig*(8)

NAME

u9fs − serve 9P from Unix

SYNOPSIS

`u9fs` [*directory*]

DESCRIPTION

U9fs is *not* a Plan 9 program. Instead it is a program that serves Unix files to Plan 9 machines using the 9P protocol (see *intro*(5)). It is to be invoked on a Unix machine by `inetd` with its standard input, output, and error connected to a network connection, typically TCP on an Ethernet. It runs as user `root` and multiplexes access to multiple Plan 9 clients over the single wire by simulating Unix permissions itself.

If a *directory* is specified *u9fs* first does a Unix `chroot` system call to that directory.

Plan 9 calls this service `9fs` with TCP service number 17008 on the Ethernet. Set up this way on a machine called, say, `kremvax`, *u9fs* may be connected to the name space of a Plan 9 process by

```
9fs kremvax
```

Due to a bug in some versions of the IP software, some systems will not accept the service name `9fs`, thinking it a service number because of the initial digit. If so, run the service as `u9fs` or `564` and do the `srv` and `mount` by hand:

```
srv tcp!kremvax!u9fs
mount -c /srv/tcp!kremvax!u9fs /n/kremvax
```

For more information on this procedure, see *srv*(4) and *bind*(1).

U9fs serves the entire file system of the Unix machine. It forbids access to devices because the program is single-threaded and may block unpredictably. Using the `attach` specifier `device` connects to a file system identical to the usual system except it permits device access (and may block unpredictably):

```
srv tcp!kremvax!9fs
mount -c /srv/tcp!kremvax!9fs /n/kremvax device
```

(The `9fs` command does not accept an attach specifier.) Even so, device access may produce unpredictable results if the block size of the device is greater than 8192, the maximum data size of a 9P message.

The source to *u9fs* is in the Plan 9 directory `/sys/src/cmd/unix/u9fs`. To install *u9fs* on a Unix system, copy the source to a directory on that system. Edit the `makefile` to set `LOG` to a proper place for a log file and to set compile-time configuration correctly. Then compile with an ANSI C compiler and install in `/usr/etc/u9fs`. Install this line in `inetd.conf`:

```
9fs     stream  tcp     nowait  root    /usr/etc/u9fs   u9fs
```

and this in `services`:

```
9fs     564/tcp        9fs  # Plan 9 fs
```

DIAGNOSTICS

Problems are reported to `/tmp/u9fs.log`. A compile-time flag enables chatty debugging.

SEE ALSO

bind(1), *srv*(4), *ip*(3)

BUGS

The implementation of devices is unsatisfactory.

NAME

intro − introduction to the Plan 9 File Protocol, 9P

SYNOPSIS

`#include <fcall.h>`

DESCRIPTION

A Plan 9 *server* is an agent that provides one or more hierarchical file systems — file trees — that may be accessed by Plan 9 processes. A server responds to requests by *clients* to navigate the hierarchy, and to create, remove, read, and write files. The prototypical server is a separate machine that stores large numbers of user files on permanent media; such a machine is called, somewhat confusingly, a *file server.* Another possibility for a server is to synthesize files on demand, perhaps based on information on data structures inside the kernel; the *proc*(3) *kernel device* is a part of the Plan 9 kernel that does this. User programs can also act as servers.

A *connection* to a server is a bidirectional communication path from the client to the server. There may be a single client or multiple clients sharing the same connection. A server's file tree is attached to a process group's name space by *bind*(2) and *mount* calls; see *intro*(2). Processes in the group are then clients of the servers: system calls operating on files are translated into requests and responses transmitted on the connection to the appropriate service.

The *Plan 9 File Protocol*, 9P, is used for messages between *clients* and *servers*. A client transmits *requests* (*T-messages*) to a server, which subsequently returns *replies* (*R-messages*) to the client. The combined acts of transmitting (receiving) a request of a particular type, and receiving (transmitting) its reply is called a *transaction* of that type.

Each message consists of a sequence of bytes. The first byte is the message type, one of the constants in the enumeration in the include file `<fcall.h>`. The remaining bytes are parameters. Each parameter consists of a fixed number of bytes (except the *data* fields of write requests or read replies); in the message descriptions below, the number of bytes in a field is given in brackets after the field name. The two-, four-, and eight-byte fields may hold unsigned integers represented in little-endian order (least significant byte first). Fields that contain names are 28-character strings (including a terminal NUL (zero) byte). Other than the NUL terminator, all characters are legal in file names. (Systems may choose to reduce the set of legal characters to reduce syntactic problems, for example to remove slashes from name components, but the protocol has no such restriction. Plan 9 names may contain any printable character except slash and blank.) Messages are transported in byte form to allow for machine independence; *fcall*(2) describes routines that convert to and from this form into a machine-dependent C structure.

MESSAGES

`Tnop`	*tag*[2]
`Rnop`	*tag*[2]
`Tsession`	*tag*[2]
`Rsession`	*tag*[2]
`Rerror`	*tag*[2] *ename*[64]
`Tflush`	*tag*[2] *oldtag*[2]
`Rflush`	*tag*[2]
`Tauth`	*tag*[2] *fid*[2] *uid*[28] *chal*[36]
`Rauth`	*tag*[2] *fid*[2] *chal*[30]
`Tattach`	*tag*[2] *fid*[2] *uid*[28] *aname*[28] *auth*[28]
`Rattach`	*tag*[2] *fid*[2] *qid*[8]
`Tclone`	*tag*[2] *fid*[2] *newfid*[2]
`Rclone`	*tag*[2] *fid*[2]
`Tclwalk`	*tag*[2] *fid*[2] *newfid*[2] *name*[28]
`Rclwalk`	*tag*[2] *fid*[2] *qid*[8]

`Twalk`	*tag*[2] *fid*[2] *name*[28]
`Rwalk`	*tag*[2] *fid*[2] *qid*[8]
`Topen`	*tag*[2] *fid*[2] *mode*[1]
`Ropen`	*tag*[2] *fid*[2] *qid*[8]
`Tcreate`	*tag*[2] *fid*[2] *name*[28] *perm*[4] *mode*[1]
`Rcreate`	*tag*[2] *fid*[2] *qid*[8]
`Tread`	*tag*[2] *fid*[2] *offset*[8] *count*[2]
`Rread`	*tag*[2] *fid*[2] *count*[2] *pad*[1] *data*[*count*]
`Twrite`	*tag*[2] *fid*[2] *offset*[8] *count*[2] *pad*[1] *data*[*count*]
`Rwrite`	*tag*[2] *fid*[2] *count*[2]
`Tclunk`	*tag*[2] *fid*[2]
`Rclunk`	*tag*[2] *fid*[2]
`Tremove`	*tag*[2] *fid*[2]
`Rremove`	*tag*[2] *fid*[2]
`Tstat`	*tag*[2] *fid*[2]
`Rstat`	*tag*[2] *fid*[2] *stat*[116]
`Twstat`	*tag*[2] *fid*[2] *stat*[116]
`Rwstat`	*tag*[2] *fid*[2]

Each T-message has a *tag* field, chosen and used by the client to identify the message. The reply to the message will have the same tag. Clients must arrange that no two outstanding messages on the same connection have the same tag. An exception is the tag `0xFFFF`, meaning 'no tag': the client can use it, when establishing a connection, to override tag matching in `nop` and `session` messages.

The type of an R-message will either be one greater than the type of the corresponding T-message or `Rerror`, indicating that the request failed. In the latter case, the *ename* field contains a string describing the reason for failure.

The `nop` message request has no obvious effect. Its main purpose is in debugging the connection between a client and a server. It is never necessary. A `session` request initializes a connection and aborts all outstanding I/O on the connection. The set of messages between `session` requests is called a *session*.

Most T-messages contain a *fid,* a 16-bit unsigned integer that the client uses to identify a ''current file'' on the server. Fids are somewhat like file descriptors in a user process, but they are not restricted to files open for I/O: directories being examined, files being accessed by *stat*(2) calls, and so on — all files being manipulated by the operating system — are identified by fids. Fids are chosen by the client. All requests on a connection share the same fid space; when several clients share a connection, the agent managing the sharing must arrange that no two clients choose the same fids.

The first fid supplied (in an `attach` message) will be taken by the server to refer to the root of the served file tree. The `attach` identifies the user to the server and may specify a particular file tree served by the server (for those that supply more than one). A `walk` message causes the server to change the current file associated with a fid to be a file in the directory that is the old current file. Usually, a client maintains a fid for the root, and navigates by `walks` on a fid `cloned` from the root fid.

A client can send multiple T-messages without waiting for the corresponding R-messages, but all outstanding T-messages must specify different tags. The server may delay the response to a request on one fid and respond to later requests on other fids; this is sometimes necessary, for example when the client reads from a file that the server synthesizes from external events such as keyboard characters.

Replies (R-messages) to `attach`, `walk`, `open` and `create` requests convey a *qid* field back to the client. The qid represents the server's unique identification for the file being accessed: two files on the same server hierarchy are the same if and only if their qids are the same. (The client may have multiple fids pointing to a single file on a server and hence having a single qid.) The eight-byte qid fields represent two four-byte unsigned integers: first the qid *path*, then the qid *version*. The path is an integer unique among all files in the hierarchy. If a file is deleted and recreated with the same name in the same directory, the old and new path components of the qids should be different. Directories always have the `CHDIR` bit (`0x80000000`) set in their qid path. The version is a version number for a file; typically, it is incremented every time the file is modified.

An existing file can be `opened,` or a new file may be `created` in the current (directory) file. I/O of a given number of bytes (limited to 8192) at a given offset on an open file is done by `read` and `write`.

A client should `clunk` any fid that is no longer needed. The `remove` transaction deletes files.

The `stat` request returns information about the file. The *stat* field in the reply includes the file's name, access permissions (read, write and execute for owner, group and public), access and modification times, and owner and group identifications (see *stat*(2)). The owner and group identifications are 28-byte names. The `wstat` transaction allows some of a file's properties to be changed.

A request can be aborted with a `Tflush` request. When a server receives a `Tflush`, it should not reply to the message with tag *oldtag* (unless it has already replied), and it should immediately send an `Rflush`. The client should ignore replies with tag *oldtag* until it gets the `Rflush`, at which point *oldtag* may be reused.

Most programs do not see the 9P protocol directly; instead calls to library routines that access files are translated by the mount driver, *mnt*(3), into 9P messages.

DIRECTORIES

Directories are created by `create` with `CHDIR` set in the permissions argument (see *stat*(5)). The members of a directory can be found with *read*(5). All directories must support `walks` to the directory `..` (dot-dot) meaning parent directory, although by convention directories contain no explicit entry for `..` or `.` (dot). The parent of the root directory of a server's tree is itself.

ACCESS PERMISSIONS

Each server maintains a set of user and group names. Each user can be a member of any number of groups. Each group has a *group leader* who has special privileges (see *stat*(5) and *users*(6)). Every file request has an implicit user id (copied from the original `attach`) and an implicit set of groups (every group of which the user is a member).

Each file has an associated *owner* and *group* id and three sets of permissions: those of the owner, those of the group, and those of ''other'' users. When the owner attempts to do something to a file, the owner, group, and other permissions are consulted, and if any of them grant the requested permission, the operation is allowed. For someone who is not the owner, but is a member of the file's group, the group and other permissions are consulted. For everyone else, the other permissions are used. Each set of permissions says whether reading is allowed, whether writing is allowed, and whether executing is allowed. A `walk` in a directory is regarded as executing the directory, not reading it. Permissions are kept in the low-order bits of the file *mode*: owner read/write/execute permission represented as 1 in bits 8, 7, and 6 respectively (using 0 to number the low order). The group permissions are in bits 5, 4, and 3, and the other permissions are in bits 2, 1, and 0.

The file *mode* contains some additional attributes besides the permissions. If bit 31 is set, the file is a directory; if bit 30 is set, the file is append-only (offset is ignored in writes); if bit 29 is set, the file is exclusive-use (only one client may have it open at a time).

NAME

attach, session, nop − messages to initiate activity

SYNOPSIS

`Tnop`	*tag*[2]
`Rnop`	*tag*[2]
`Tsession`	*tag*[2]
`Rsession`	*tag*[2]
`Tattach`	*tag*[2] *fid*[2] *uid*[28] *aname*[28] *auth*[28]
`Rattach`	*tag*[2] *fid*[2] *qid*[8]

DESCRIPTION

The `nop` request does nothing overt but may be used to synchronize the channel between two service hosts initially.

The `session` request is used to initialize a connection between a client and a server. All outstanding I/O on the connection is aborted. The set of messages between `session` requests is called a *session*. Tags and fids must be unique per session.

The *tag* should be `NOTAG` (value `0xFFFF`) for a `nop` or `session` message.

The `attach` message serves as a fresh introduction from a user on the client machine to a server. The message identifies the user (*uid*) and may select the file tree to access (*aname*). The *auth* argument contains authorization data derived from the *chal* field of an `auth` message; see *auth*(5) and *auth*(6).

As a result of the `attach` transaction, the client will have a connection to the root directory of the desired file tree, represented by *fid*. An error is returned if *fid* is already in use. The server's idea of the root of the file tree is represented by the returned `qid`.

ENTRY POINTS

An `attach` transaction will be generated for kernel devices (see *intro*(3)) when a system call evaluates a file name beginning with `#`. *Pipe*(2) generates an attach on the kernel device *pipe*(3). `Mount` (see *bind*(2)) generates `auth` and `attach` messages to the remote file server. When the kernel boots, an *attach* is made to the root device, *root*(3), and then an `auth` and an `attach` are made to the requested file server machine.

SEE ALSO

auth(5), *auth*(6)

NAME

auth − file system authentication

SYNOPSIS

`Tauth` *tag*[2] *fid*[2] *uid*[28] *chal*[36]
`Rauth` *tag*[2] *fid*[2] *chal*[30]

DESCRIPTION

The `auth` message is used to authorize a connection. It is issued before an `attach`. *Fid* and *uid* are the same as for `attach`.

The *chal* field of a `Tauth` message contains a 36-byte string encrypted with the client's authentication key. The (decrypted) string contains a byte with value 1, a seven byte client challenge, and the server's name NUL-padded to 28 (`NAMELEN`) bytes.

The *chal* field of the `Rauth` reply message is also encrypted with the client's key. The decrypted string contains a byte with value 4, the client's challenge, a seven byte *ticket key,* and a fifteen byte *ticket.* The ticket is placed in the *auth* field of a subsequent `attach` message to validate a connection.

The ticket key is currently unused. It may one day be used to encrypt subsequent communication with the server.

These messages are also documented in the section of *auth*(6) describing the *fsauth* protocol.

If a server does not perform authentication, it should return an `Rerror` when it receives an `auth`.

ENTRY POINTS

Mount (see *bind*(2)) generates an `auth` transaction to the remote file server. When the kernel boots, an `auth` is made to the requested file server machine.

SEE ALSO

auth(6)

NAME

clone − duplicate a fid

SYNOPSIS

`Tclone` *tag*[2] *fid*[2] *newfid*[2]
`Rclone` *tag*[2] *fid*[2]

DESCRIPTION

The `clone` request carries as arguments an existing *fid* and a proposed *newfid* (which must not be in use) that the client wishes to associate with the same file as *fid*. The *fid* must be valid in the current session and must not have been opened for I/O by an `open` or `create` message. After a successful `clone` and before any subsequent messages, *fid* and *newfid* are indistinguishable.

ENTRY POINTS

A `clone` message is generated by any system call that evaluates a path name and by a *read* of a union directory.

NAME

clunk − forget about a fid

SYNOPSIS

`Tclunk` *tag*[2] *fid*[2]
`Rclunk` *tag*[2] *fid*[2]

DESCRIPTION

The `clunk` request informs the file server that the current file represented by *fid* is no longer needed by the client. The actual file is not removed on the server.

Once a fid has been clunked, the same fid can be reused in a new `clone` request.

Even if the `clunk` returns an error, the *fid* is no longer valid.

ENTRY POINTS

A `clunk` message is generated by *close* and indirectly by other actions such as failed *open* calls.

NAME

clwalk − clone, then search a directory, and change to a file within it

SYNOPSIS

`Tclwalk` *tag*[2] *fid*[2] *newfid*[2] *name*[28]
`Rclwalk` *tag*[2] *fid*[2] *qid*[8]

DESCRIPTION

The `clwalk` request is a combination of a `clone` request (see *clone*(5)) followed by a `walk` request (see *walk*(5)) on the new *fid*. If the walk fails, there is an implicit `clunk` of `newfid`.

ENTRY POINTS

The `clwalk` message is an optimization for use on low-speed lines; it is not generated by the kernel. The *cfs*(4) cached file system generates it as a side-effect of any system call that interprets a file name.

NAME

error − return an error

SYNOPSIS

`Rerror` *tag*[2] *ename*[28]

DESCRIPTION

The `Rerror` request (there is no `Terror`) is used to return an error string describing the failure of a transaction. It replaces the corresponding reply message that would accompany a successful call; its tag is that of the request.

NAME

flush − abort a message

SYNOPSIS

`Tflush` *tag*[2] *oldtag*[2]
`Rflush` *tag*[2]

DESCRIPTION

When the response to a request is no longer needed, such as when a user interrupts a process doing a *read*(2), a `Tflush` request is sent to the server to purge the pending response. The message being flushed is identified by *oldtag.* The semantics of `flush` depends on messages arriving in order.

The server must answer the `flush` message immediately. If it recognizes *oldtag* as the tag of a pending transaction, it should abort any pending response and discard that tag. In either case, it should respond with an `Rflush` echoing the tag (not *oldtag*) of the `Tflush` message. A `Tflush` can never be responded to by an `Rerror` message.

When the client sends the `Tflush`, it should disregard all messages received with tag *oldtag* until the corresponding `Rflush` is received, at which point *oldtag* may be recycled for subsequent messages.

Several exceptional conditions are handled correctly by the above specification: sending multiple flushes for a single tag, flushing a `Tflush`, and flushing an invalid tag.

NAME

open, create − prepare a fid for I/O on an existing or new file

SYNOPSIS

`Topen` *tag*[2] *fid*[2] *mode*[1]
`Ropen` *tag*[2] *fid*[2] *qid*[8]

`Tcreate` *tag*[2] *fid*[2] *name*[28] *perm*[4] *mode*[1]
`Rcreate` *tag*[2] *fid*[2] *qid*[8]

DESCRIPTION

The `open` request asks the file server to check permissions and prepare a fid for I/O with subsequent `read` and `write` messages. The *mode* field determines the type of I/O: 0, 1, 2, and 3 mean *read access, write access, read and write access,* and *execute access,* to be checked against the permissions for the file. In addition, if *mode* has the `OTRUNC` (`0x10`) bit set, the file is to be truncated, which requires write permission; if the *mode* has the `ORCLOSE` (`0x40`) bit set, the file is to be removed when the fid is clunked, which requires permission to remove the file from its directory. If other bits are set in *mode* they will be ignored. It is illegal to write a directory, truncate it, or attempt to remove it on close. If the file is marked for exclusive use (see *stat*(5)), only one client can have the file open at any time. That is, after such a file has been opened, no other open will succeed until *fid* has been clunked. All these permissions are checked at the time of the `open` request; subsequent changes to the permissions of files do not affect the ability to read, write, or remove an open file.

The `create` request asks the file server to create a new file with the *name* supplied, in the directory (*dir*) represented by *fid*, and requires write permission in the directory. The owner of the file is the implied user id of the request, the group of the file is the same as *dir*, and the permissions are the value of

```
(perm&(~0777|0111)) | (dir.perm&perm&0666)
```

if a regular file is being created and

```
(perm&~0777) | (dir.perm&perm&0777)
```

if a directory is being created. This means, for example, that if the *create* allows read permission to others, but the containing directory does not, then the created file will not allow others to read the file.

Finally, the newly created file is opened according to *mode*, and *fid* will represent the newly opened file. *Mode* is not checked against the permissions in *perm.* The *qid* for the new file is returned with the `create` response.

Directories are created by setting the `CHDIR` bit (`0x80000000`) in the *mode*.

The names `.` and `..` are special; it is illegal to create files with these names.

It is an error for either of these messages if the fid is already the product of a successful `open` or `create` message.

An attempt to `create` a file in a directory where the given *name* already exists will be rejected; in this case, *create* (see *open*(2)) uses `open` with truncation. The algorithm used by *create* is: first walk to the directory to contain the file. If that fails, return an error. Next `walk` to the specified file. If the `walk` succeeds, send a request to `open` and truncate the file and return the result, successful or not. If the `walk` fails, send a create message. If that fails, it may be because the file was created by another process after the previous walk failed, so (once) try the `walk` and `open` again. For the behavior of *create* on a union directory, see *bind*(2).

ENTRY POINTS

Open and *create* both generate `open` messages; only *create* generates a `create` message.

NAME

read, write − transfer data from and to a file

SYNOPSIS

`Tread` *tag*[2] *fid*[2] *offset*[8] *count*[2]
`Rread` *tag*[2] *fid*[2] *count*[2] *pad*[1] *data*[*count*]

`Twrite` *tag*[2] *fid*[2] *offset*[8] *count*[2] *pad*[1] *data*[*count*]
`Rwrite` *tag*[2] *fid*[2] *count*[2]

DESCRIPTION

The `read` request asks for *count* bytes of data from the file identified by *fid,* which must be opened for reading, starting *offset* bytes after the beginning of the file. *Count* must be less than or equal to `MAXFDATA` (8192, defined in `<fcall.h>`). The bytes are returned with the `read` reply message.

The *count* field in the reply indicates the number of characters returned. This may be less than the requested amount. If the *offset* field is greater than the number of characters in the file, a count of zero will be returned. For directories, `read` returns an integral number of directory entries exactly as in `stat` (see *stat*(5)), one for each member of the directory. The read request message must have `offset` and `count` zero modulo `DIRLEN`.

The `write` request asks that *count* bytes of data be recorded in the file identified by *fid,* which must be opened for writing, starting *offset* characters after the beginning of the file. If the file has been opened append only, the data will be placed at the end of the file regardless of *offset*. Directories may not be written.

The `write` reply records the number of characters actually written. It is usually an error if this is not the same as requested.

ENTRY POINTS

`Read` and `write` messages are generated by the corresponding calls. Because of the `MAXFDATA` limit, more than one message may be produced by a single call.

NAME

remove − remove a file from a server

SYNOPSIS

`Tremove` *tag*[2] *fid*[2]
`Rremove` *tag*[2] *fid*[2]

DESCRIPTION

The `remove` request asks the file server both to remove the file represented by *fid* and to `clunk` the *fid,* even if the remove fails. This request will fail if the client does not have write permission in the parent directory.

It is correct to consider `remove` to be a `clunk` with the side effect of removing the file if permissions allow.

ENTRY POINTS

`Remove` messages are generated by *remove.*

NAME

stat, wstat − inquire or change file attributes

SYNOPSIS

`Tstat` *tag*[2] *fid*[2]
`Rstat` *tag*[2] *fid*[2] *stat*[116]

`Twstat` *tag*[2] *fid*[2] *stat*[116]
`Rwstat` *tag*[2] *fid*[2]

DESCRIPTION

The `stat` transaction inquires about the file identified by *fid*. The reply will contain a 116-byte (`DIRLEN` in `<libc.h>`) machine-independent *directory entry* laid out as follows:

name[28]	file name
uid[28]	owner name
gid[28]	group name
qid.path[4]	the file server's identification for the file
qid.vers[4]	version number for given path
mode[4]	permissions and flags
atime[4]	last access time
mtime[4]	last modification time
length[8]	length of file in bytes
type[2]	for kernel use
dev[2]	for kernel use

Integers in this encoding are in little-endian order (least significant byte first). The *convM2D* and *convD2M* routines (see *fcall*(2)) convert between directory entries and C `structs`.

This encoding may be turned into a machine dependent *Dir* structure (see *stat*(2)) using routines defined in *fcall*(2).

The *mode* contains permission bits as described in *intro*(5) and the following: `0x80000000` (this file is a directory), `0x40000000` (append only), `0x20000000` (exclusive use). Writes to append-only files always place their data at the end of the file; the *offset* in the `read` or `write` message is ignored. Exclusive use files may be open for I/O by only one fid at a time across all clients of the server. If a second open is attempted, it draws an error. Servers may implement a timeout on the lock on an exclusive use file: if the fid holding the file open has been unused for an extended period (of order at least minutes), it is reasonable to break the lock and deny the initial fid further I/O.

The two time fields are measured in seconds since the epoch (Jan 1 00:00 1970 local time). The *mtime* field reflects the time of the last change of content. For a plain file, *mtime* is the time of the most recent `create`, `open` with truncation, or `write`; for a directory it is the time of the most recent `remove`, `create`, or `wstat` of a file in the directory. Similarly, the *atime* field records the last `read` of the contents; also it is set whenever *mtime* is set. In addition, for a directory, it is set by an `attach`, `walk`, or `create`, all whether successful or not.

The *length* records the number of bytes in the file. Directories and most files representing devices have a conventional length of 0.

The `stat` request requires no special permissions.

The `wstat` request can change some of the file status information. The *name* can be changed by anyone with write permission in the parent directory; it is an error to change the name to that of an existing file. The *mode* can be changed by the owner of the file or the group leader of the file's current group. The directory bit cannot be changed by a `wstat`; the other defined permission and mode bits can. The *gid* can be changed: by the owner if also a member of the new group; or by the group leader of the file's current group if also leader of the new group (see *intro*(5) for more information about permissions and *users*(6) for users and groups). None of the other data can be altered by a `wstat`. In particular, there is no way to change the owner of a file.

A *read* of a directory yields an integral number of directory entries in the machine independent encoding given above (see *read*(5)).

ENTRY POINTS

`Stat` messages are generated by *fstat* and *stat.*

`Wstat` messages are generated by *fwstat* and *wstat.*

NAME

walk − descend a directory hierarchy

SYNOPSIS

`Twalk` *tag*[2] *fid*[2] *name*[28]
`Rwalk` *tag*[2] *fid*[2] *qid*[8]

DESCRIPTION

The `walk` request looks for the file *name* in the directory represented by *fid*.

For the `walk` to succeed, the file identified by *fid* must be a directory, and the implied user of the request must have permission to search the directory (see *intro*(5)).

After a successful `walk`, *fid* represents the specified file. The *qid* for the new file is returned with the `walk` response.

ENTRY POINTS

A call to *chdir*(2) causes a `walk`. One or more `walk` messages may be generated by any of the following calls, which evaluate file names: *bind*, *create*, *mount*, *open*, *stat*, *wstat*. The file name element **.** (dot) is interpreted locally and is not transmitted in `walk` messages.

NAME

intro − introduction to file formats

DESCRIPTION

This section of the manual describes file formats and other miscellanea such as *troff* macro packages.

NAME

a.out − object file format

SYNOPSIS

`#include <a.out.h>`

DESCRIPTION

An executable Plan 9 binary file has six sections: a header, the program text, the data, a symbol table, an PC/SP offset table, and finally a PC/line number table. The header format, given in `<a.out.h>`, contains 4-byte integers in big-endian order:

```
typedef struct Exec Exec;
struct Exec {
         long       magic;      /* magic number */
         long       text;       /* size of text segment */
         long       data;       /* size of initialized data */
         long       bss;        /* size of uninitialized data */
         long       syms;       /* size of symbol table */
         long       entry;      /* entry point */
         long       spsz;       /* size of pc/sp offset table */
         long       pcsz;       /* size of pc/line number table */
};
#define  _MAGIC(b)  ((((4*b)+0)*b)+7)
#define  A_MAGIC    _MAGIC(8)   /* vax */
#define  Z_MAGIC    _MAGIC(10)  /* hobbit */
#define  I_MAGIC    _MAGIC(11)  /* intel 386 */
#define  J_MAGIC    _MAGIC(12)  /* intel 960 */
#define  K_MAGIC    _MAGIC(13)  /* sparc */
#define  P_MAGIC    _MAGIC(14)  /* hp-pa */
#define  V_MAGIC    _MAGIC(16)  /* mips 3000 */
```

Sizes are expressed in bytes. The size of the header is not included in any of the other sizes.

When a Plan 9 binary file is executed, a memory image of three segments is set up: the text segment, the data segment, and a stack. The text segment begins at virtual address *p*, a multiple of the machine-dependent page size. The text segment consists of the header and the first `text` bytes of the binary file. The `entry` field gives the virtual address of the start of the program, usually *p*+`sizeof(struct exec)`. The data segment starts at the first page-rounded virtual address after the text segment. It consists of the next `data` bytes of the binary file, followed by `bss` bytes initialized to zero. The stack occupies the highest possible locations in the core image, automatically growing downwards. The data segment may be extended by *brk*(2).

The next `syms` (possibly zero) bytes of the binary file contain symbol table entries. The layout of a symbol table entry, also in big-endian order, is also in `<a.out.h>`:

```
#define   NNAME   20
typedef   struct  Sym Sym;
struct    Sym {
          long    value;
          char    type;
          char    name[NNAME];   /* NUL-terminated */
          char    pad[3];
};
```

The `type` field is one of the following characters:

`T`	text segment symbol
`t`	static text segment symbol

`L`	leaf function text segment symbol
`l`	static leaf function text segment symbol
`D`	data segment symbol
`d`	static data segment symbol
`B`	bss segment symbol
`b`	static bss segment symbol
`a`	automatic (local) variable symbol
`p`	function parameter symbol

A few others are described below. The symbols in the symbol table appear in the same order as the program components they describe.

After the symbol table comes a `spsz`-byte SP offset table and a `pcsz`-byte source code line number table. Both tables may be empty. The Plan 9 compilers implement a virtual frame pointer rather than dedicating a register; moreover, on the MC68020 and i386 there is a variable offset between the stack pointer and the frame pointer. The PC/SP offset table encodes this offset as a function of program location.

The table is encoded as a byte stream. By interpreting the stream setting the PC to the base of the text segment and the offset to zero, the offset can be computed for any PC. A byte value of 0 is followed by four bytes that hold, in big-endian order, a constant to be added to the offset. A byte value of 1 to 64 is multiplied by four and added, without sign extension, to the offset. A byte value of 65 to 128 is reduced by 64, multiplied by four, and subtracted from the offset. A byte value of 129 to 255 is reduced by 129, multiplied by the quantum of instruction size (e.g. four bytes on the RISC machines, two on the MC68020, one on the i386), and added to the current PC without changing the offset. After any of these operations, the instruction quantum is added to the PC.

The same algorithm may be run over the PC/line number table to recover the absolute source line number from a given program location. The absolute line number (starting from zero) counts the newlines in the C-preprocessed source seen by the compiler. Three symbol types in the main symbol table facilitate conversion of the absolute number to source file and line number:

`f`	source file name components
`z`	source file name
`Z`	source file line offset

The `f` symbol identifies an integer (the value of the 'symbol') to represent a unique file path name component (the name of the 'symbol'). These path components are used by the `z` symbol to represent a file name: the first byte of the name field is always 0; the remaining `NNAME-1` bytes hold a zero-terminated array of 16-bit values (in big-endian order) that represent file name components from `f` symbols. These components, when separated by slashes, form a file name. The initial slash of a file name is recorded in the symbol table by an `f` symbol; when forming file names from `z` symbols an initial slash is not to be assumed.

The `z` symbols are clustered, one set for each object file from which the program was assembled, before any text symbols from that object file. The set of `z` symbols for an object file form a *history stack* of the included source files from which the object file was compiled. The value associated with each `z` symbol is the absolute line number at which that file was included in the source; if the name associated with the `z` symbol is null, the symbol represents the end of an included file, that is, a pop of the history stack. If the value of the `z` symbol is 1 (one), it represents the start of a new history stack.

To recover the source file and line number for a program location, find the text symbol containing the location and then the first history stack preceding the text symbol in the symbol table. Next, interpret the PC/line offset table to discover the absolute line number for the program location. Using the line number, scan the history stack to find the set of source files open at that location. The line number within the file can be found using the line numbers in the history stack.

The `Z` symbols correspond to `#line` directives in the source; they specify an adjustment to the line number to be printed by the above algorithm. The offset is associated with the first previous `z` symbol in the symbol table.

SEE ALSO

db(1), *2a*(1), *2l*(1), *nm*(1), *strip*(1)

BUGS

There is no type information in the symbol table.

NAME

ar − archive (library) file format

SYNOPSIS

`#include <ar.h>`

DESCRIPTION

The archive command *ar*(1) is used to combine several files into one. Archives are used mainly as libraries to be searched by the loaders *2l*(1) et al.

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:

```
#define ARMAG   "!<arch>\n"
#define SARMAG  8

#define ARFMAG  "`\n"

struct ar_hdr {
        char    name[16];
        char    date[12];
        char    uid[6];
        char    gid[6];
        char    mode[8];
        char    size[10];
        char    fmag[2];
};
#define SAR_HDR 60
```

The name is a blank-padded string. The `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 `mode`, which is octal. The date is the modification date of the file (see *stat*(2)) at the time of its insertion into the archive. The mode is the low 9 bits of the file permission mode, in octal. The length of the header is `SAR_HDR`. Because `struct ar_hdr` may be padded on some machines, `SAR_HDR` should be used in preference to `sizeof(struct ar_hdr)` when reading and writing file headers.

Each file begins on an even (0 mod 2) boundary; a newline is inserted between files if necessary. Nevertheless `size` reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

SEE ALSO

ar(1), *2l*(1), *nm*(1), *stat*(2)

BUGS

The `uid` and `gid` fields are unused in Plan 9. They provide compatibility with Unix *ar* format.

NAME

fsauth, rexauth, chal, changekey − authentication services

DESCRIPTION

This manual page describes the *authentication services*: the protocols used to authorize connections, confirm the identities of users and machines, and maintain the associated databases. The machine that provides these services is called the *authentication server* and may be a stand-alone machine or a general-use machine such as a CPU server. The network database holds for each public machine, such as a CPU server or file server, the name of the authentication server that machine uses.

There are four authentication services. Each is executed by making a network call from the machine wishing authentication to the authentication server and exchanging messages. The services are:

fsauth	authenticate file system attaches
rexauth	authenticate remote execution from a Plan 9 machine
chal	authenticate connections from a non-Plan 9 machine using a SecureNet box (see *securenet*(8))
changekey	change the key for a user or client.

Multiple *fsauth* requests may be processed on a single connection to the authentication server. The other protocols accept only one request per call.

When a client calls another machine, say a file server, using the 9P protocol, the file server receives a `Tauth` message containing information about the user making the call (see *auth*(5)). The file server exchanges some messages with the authentication server using the *fsauth* protocol described below. It then returns an `Rauth` message to the client containing a *ticket* to be used by the client in the subsequent `Tattach` message (see *attach*(5)); that ticket guarantees that the user requesting the service is the one validated by the authentication server.

In describing the protocols, the following notation is used.

A	The authentication server.
S	A CPU server or file server.
C	A client connecting to S. When any of these appears as part of a message, it refers to the textual name of the agent padded with zeros to a total of `NAMELEN` bytes.
K*x*	The seven byte authentication key of *x*; *x* is either S or C. Servers keep a private copy of their keys, typically in non-volatile RAM, and encrypt using the library functions *encrypt*(2) and *decrypt.* Clients keep a copy of the current user's key in the file `#c/key` and encrypt using the file `#c/crypt` (see *cons*(3)).
K'C	C's *network key,* stored in C's SecureNet box. Encryption with K'C is done with the algorithm described in *securenet*(8). KC may also be used in place of K'C to execute the *chal* protocol without a SecureNet box; in this case, the *netcrypt* routine is used for encryption. In either case the result of the encryption is a variable length text string, to be transmitted with its terminating NUL.
KT	A *ticket key,* a random number stored in a ticket.
*x*PC	A password for the client: a 10 byte NUL-terminated string. The character *x* is either *o* for an old password or *n* for a new one.
Ch*x*	A seven byte challenge made by *x*; *x* is one of A, S, or C.
NetCh	A NUL-terminated string of between 1 and 6 digits for encryption using K'C. NetCh is a challenge generated by A and is transmitted as a variable length NUL-terminated string.
Kx{s}	Braces denote encryption. Kx{s} is the result of encrypting *s* using key Kx.
E	An error message `ERRLEN` bytes long.

Arrows indicate communication. The authentication server communicates only with a server, so a communication between A and C indicates that S forwards the message uninterpreted.

Consider the *fsauth* protocol to validate a connection to a file server. Here is the concise notation of the protocol; following that is a prose description of its execution:

Fsauth

[1] C→S KC{FScchal, ChC, S}, C
[2] S→A KS{FSschal, ChS, C, KC{FScchal, ChC, S}}, S

[3] A→S KS{FSok, ChS, KC{FSctick, ChC, KT, KS{FSstick, ChS, KT}}}
or
[4] A→S KS{FSerr, ChS, E}
[5] S→C KC{FSctick, ChC, KT, KS{FSstick, ChS, KT}}
[6] C→S KS{FSstick, ChS, KT}

[1] The client prepares a string containing an initial byte with value `FScchal` (defined in `<auth.h>`), a seven-byte random string, ChC, and the name of the server it is calling, e.g. `kremvax`, padded with zeros to `NAMELEN` bytes, for a total of 1+7+`NAMELEN`=36 bytes. If the client does not care which file system it attaches to, it can substitute the string `any` for the name of the server. It calls *encrypt*(2) to encrypt this string using the password typed by the user at login time and stored in `#c/key` (KC). Next the client prepares a `Tauth` message (see *auth*(5)): `chal` is set to the result of the encryption (KC{FScchal,ChC,S}) and `uid` to the name of the user placing the call (C). This message is transmitted to the server, S.

[2] The server prepares a string containing an initial byte with value `FSschal`, another 7-byte random string (ChS), the name of the client (C), and the contents of the `chal` field of the `Tauth` message. It encrypts this using the server's key (KS) and appends its own name to the 2*36=72 resulting bytes and sends the total 72+`NAMELEN`=100 bytes to the authentication server.

The authentication server responds with one of two results, both encrypted with the server's key. [3] If the authentication is approved, the (decrypted) result contains a byte with value `FSok`, the server's challenge (ChS), and a thirty-byte string, called *chal,* encrypted with the client's key, to be returned to the client (KC{FSctick,ChC,KT,KS{FSstick,ChS,KT}}). [4] If the authentication is not approved, the result contains a byte with value FSerr, the server challenge, and an error message.

[5] The server decrypts the response and sends either an `Rauth` message with the `chal` field set to the *chal* string or an `Rerror` message containing the error describing why authentication failed. (The error case is not shown in the concise form; it is outside the authentication protocol.)

[6] If authentication succeeds, the client decrypts the `chal` field of the `Rauth` and extracts the 15-byte long ticket (KS{FSstick,ChS,KT}). It places that in the `auth` field of the `Tattach` message it sends to establish the connection to the server.

In the remaining protocol descriptions, the bytes transmitted in the communications are exactly as presented in the concise notation.

Rexauth
[1] S→C KS{RXschal, ChS}
[2] C→A KC{KS{RXschal, ChS}, S, RXcchal, ChC}, C
[3] A→C KC{KS{RXstick, ChS, C, KC}, RXctick, ChC}
[4] C→S KS{RXstick, ChS, C, KC}

[1] The client C calls the (CPU) server, which recognizes the incoming call and reads the already-encrypted string KS{RXschal,ChS} from the file `#c/chal` and transmits it to C. `RXschal` is a single byte identifying the message type.

[2] The client encapsulates the message in a larger message containing the server name (S) , an `RXcchal` byte, a client challenge (ChC), all encrypted, and the client name (C) (KC{KS{RXschal,ChS},S,RXcchal,ChC},C). This message is sent to S which forwards it to the authentication server A.

[3] The authentication server forms a new message (KC{KS{RXstick,ChS,C,KC},RXctick,ChC}) and sends it through the server to the client.

[4] The client decrypts this message and extracts a ticket (KS{RXstick,ChS,C,KC}) which it sends to the server. The ticket contains the client key (KC) so the server may validate further requests for the client from the server.

Chal
[1] S→A C, S, KS{RXschal, ChS}

[2] A→C NetCh
[3] C→A K'C{NetCh}
[4] A→S KS{RXstick, ChS, C, KC}

The *chal* protocol is closely related to *rexauth.* The main difference [2] is that the authentication server sends to the client a challenge (NetCh) to be encrypted by a SecureNet box. The result is returned [3] to the authentication server. The challenge and response are variable-length NUL-terminated strings of digits. The rest of the protocol is isomorphic to *rexauth.*

Changekey
[1] A→C ChA
[2] C→A C, KC{CKcchal, ChA, oPC, nPC}
[3] A→C password changed
or
[4] A→C E

This protocol is run directly between a user and the authentication server to change the key for a user; no other server is involved.

[1] The authentication server sends a challenge directly to the client.

[2] The client constructs a message containing the name (C) and an encrypted string holding a `CKcchal` byte, the challenge, the old password for the client (oPC) and the new password (nPC). It returns this to the authentication server.

[3] If the change is accepted the authentication server returns the text string `password changed`.

[4] Otherwise, it returns an error string (E).

SEE ALSO

auth(2), *encrypt*(2), *intro*(5), *auth*(5)

BUGS

The *rexauth* and *chal* protocols should create a new key for the server to hold on behalf of the client.

NAME

bitmap − external format for bitmaps

SYNOPSIS

`#include <libg.h>`

DESCRIPTION

Bitmaps are described in *graphics*(2). Fonts and bitmaps are stored in external files in machine-independent formats.

Bitmap files are read and written using *rdbitmapfile* and *wrbitmapfile* (see *balloc*(2)). A bitmap file starts with 5 decimal strings: `ldepth`, `r.min.x`, `r.min.y`, `r.max.x`, and `r.max.y`. Each number is right-justified and blank padded in 11 characters, followed by a blank. The rest of the file contains the `r.max.y`−`r.min.y` rows of bitmap data. A *row* consists of the byte containing pixel `r.min.x` and all the bytes up to and including the byte containing pixel `r.min.x`–1. A pixel with x-coordinate = x in a bitmap with `ldepth` = l will appear as $w = 2^l$ contiguous bits in a byte, with the pixel's high order bit starting at the byte's bit number $w\times((x \bmod 8)/w)$, where bits within a byte are numbered 0 to 7 from the high order to the low order bit. If w is greater than 8, it is a multiple of 8, so pixel values take up an integral number of bytes. Rows contain integral number of bytes, so there may be some unused pixels at either end of a row.

The *rdbitmap* and *wrbitmap* functions described in *balloc*(2) also deal with rows in this format, stored in user memory.

Some small images, in particular 48×48 face files as used by *seemail* (see *mail*(1)) and 16×16 cursors, are stored textually, suitable for inclusion in C source. Each line of text represents one scan line as a comma-separated sequence of hexadecimal bytes, shorts, or words in C format. For cursors, each line defines a pair of bytes. (It takes two images to define a cursor; each must be stored separately to be processed by programs such as *tweak*(1).) Face files of one bit per pixel are stored as a sequence of shorts, those of larger pixel sizes as a sequence of longs. Software that reads these files must deduce the image size from the input; there is no header. These formats reflect history rather than design.

SEE ALSO

tweak(1), *graphics*(2), *bitblt*(2), *balloc*(2), *font*(6)

NAME

font, subfont − external format for fonts and subfonts

SYNOPSIS

`#include <libg.h>`

DESCRIPTION

Fonts and subfonts are described in *cachechars*(2).

External fonts are described by a plain text file that can be read using *rdfontfile.* The format of the file is a header followed by any number of subfont range specifications. The header contains two numbers: the height and the ascent. The height is the inter-line spacing and the ascent is the distance from the top of the line to the baseline. These numbers are chosen to display consistently all the subfonts of the font. A subfont range specification contains two numbers and a file name. The numbers are the inclusive range of characters covered by the subfont, and the file name names an external file suitable for *rdsubfontfile*. The minimum number of a covered range is mapped to character zero of the corresponding subfont. If the subfont file name does not begin with a slash, it is taken relative to the directory containing the font file. Each field must be followed by some white space. Each numeric field may be C-format decimal, octal, or hexadecimal.

External subfonts are represented in a more rigid format that can be read and written using *rdsubfontfile* and *wrsubfontfile* (see *subfalloc*(2)). The format for subfont files is: a bitmap containing character images, followed by a subfont header, followed by character information. The bitmap has the format for external bitmap files described in *bitmap*(6). The subfont header has 3 decimal strings: `n`, `height`, and `ascent`. Each number is right-justified and blank padded in 11 characters, followed by a blank. The character `info` consists of `n+1` 6-byte entries, each giving the `Fontchar` `x` (2 bytes, low order byte first), `top`, `bottom`, `left`, and `width`. The `x` field of the last `Fontchar` is used to calculate the bitmap width of the previous character; the other fields in the last `Fontchar` are irrelevant.

SEE ALSO

graphics(2), *bitblt*(2), *cachechars*(2), *subfalloc*(2)

NAME

keyboard − how to type characters

DESCRIPTION

Keyboards are idiosyncratic. It should be obvious how to type ordinary ASCII characters, backspace, tab, escape, and newline. In Plan 9, the key labeled `Return` or `Enter` generates a newline (`0x0A`); if there is a key labeled `Line Feed`, it generates a carriage return (`0x0D`); Plan 9 eschews CRLFs. All control characters are typed in the usual way; in particular, control-J is a line feed and control-M a carriage return. On the Safari, the key labeled `Caps Lock` acts as an additional control key.

The delete character (`0x7F`) may be generated by a different key, one near the extreme upper right of the keyboard. On the Next it is the key labeled `*` (not the asterisk above the 8). On the SLC, delete is labeled `Num Lock` (the key above `Backspace` labeled `Delete` functions as an additional backspace key). On the other keyboards, the key labeled `Del` generates the delete character.

The view character (`0x80`), used by *8½*(1) and *sam*(1), causes windows to scroll forward. It is generally somewhere near the lower right of the main key area. The scroll character is generated by the `VIEW` key on the Gnot, the `Alt Graph` key on the SLC, and any of the three arrow keys ←, ↓, and → on the other terminals.

Characters in Plan 9 are runes (see *utf*(6)). Any 16-bit rune can be typed using a compose key followed by several other keys. The compose key is also generally near the lower right of the main key area: the `NUM PAD` key on the Gnot, the `Alternate` key on the Next, the `Compose` key on the Slc, the `Option` key on the Magnum, and either `Alt` key on the Safari. After typing the compose key, type a capital `X` and exactly four hexadecimal characters (digits and `a` to `f`) to type a single rune with the value represented by the typed number. There are shorthands for some characters. Follow the compose key with appropriate two-character sequence to generate the desired rune:

¡	!!	¢	c$	£	l$	¤	g$
¥	y$	¦	||	§	SS	¨	""
©	cO	ª	sa	«	<<	¬	no
-	--	®	rO	¯	__	°	de
±	+-	²	s2	³	s3	´	''
µ	mi	¶	pg	·	..	¸	,,
¹	s1	º	s0	»	>>	¼	14
½	12	¾	34	¿	??	À	`A
Á	'A	Â	^A	Ã	~A	Ä	"A
Å	oA	Æ	AE	Ç	,C	È	`E
É	'E	Ê	^E	Ë	"E	Ì	`I
Í	'I	Î	^I	Ï	"I	Ð	D-
Ñ	~N	Ò	`O	Ó	'O	Ô	^O
Õ	~O	Ö	"O	×	mu	Ø	/O
Ù	`U	Ú	'U	Û	^U	Ü	"U
Ý	'Y	Þ	|P	ß	ss	à	`a
á	'a	â	^a	ã	~a	ä	"a
å	oa	æ	ae	ç	,c	è	`e
é	'e	ê	^e	ë	"e	ì	`i
í	'i	î	^i	ï	"i	ð	d-
ñ	~n	ò	`o	ó	'o	ô	^o
õ	~o	ö	"o	÷	-:	ø	/o
ù	`u	ú	'u	û	^u	ü	"u
ý	'y	þ	|p	ÿ	"y	α	*a
β	*b	γ	*g	δ	*d	ε	*e
ζ	*z	η	*y	θ	*h	ι	*i
κ	*k	λ	*l	μ	*m	ν	*n

ξ	`*c`	ο	`*o`	π	`*p`	ρ	`*r`
ς	`ts`	σ	`*s`	τ	`*t`	υ	`*u`
φ	`*f`	χ	`*x`	ψ	`*q`	ω	`*w`
Α	`*A`	Β	`*B`	Γ	`*G`	Δ	`*D`
Ε	`*E`	Ζ	`*Z`	Η	`*Y`	Θ	`*H`
Ι	`*I`	Κ	`*K`	Λ	`*L`	Μ	`*M`
Ν	`*N`	Ξ	`*C`	Ο	`*O`	Π	`*P`
Ρ	`*R`	Σ	`*S`	Τ	`*T`	Υ	`*U`
Φ	`*F`	Χ	`*X`	Ψ	`*Q`	Ω	`*W`
←	`<-`	↑	`ua`	→	`->`	↓	`da`
↔	`ab`	∀	`fa`	∃	`te`	∂	`pd`
∅	`es`	∆	`De`	∇	`gr`	∉	`!m`
∍	`st`	∗	`**`	•	`bu`	√	`sr`
∝	`pt`	∞	`if`	∠	`an`	∧	`l&`
∨	`l|`	∩	`ca`	∪	`cu`	∫	`is`
∴	`tf`	≃	`~=`	≅	`cg`	≈	`~~`
≠	`!=`	≡	`==`	≤	`<=`	≥	`>=`
⊂	`sb`	⊃	`sp`	⊄	`!b`	⊆	`ib`
⊇	`ip`	⊕	`O+`		`O-`	⊗	`Ox`
	`tu`		`Tu`	◊	`lz`	…	`el`

Note the difference between ß (ss) and µ (micron) and the Greek β and μ. As well, white and black chess pieces may be escaped using the sequence color (`w` or `b`) followed by piece (`k` for king, `q` for queen, `r` for rook, `n` for knight, `b` for bishop, or `p` for pawn).

SEE ALSO

Intro(1), *ascii*(1), *tcs*(1), *8½*(1), *sam*(1), *cons*(3), *utf*(6)

NAME

man − macros to typeset manual

SYNOPSIS

`nroff -man` *file* ...

`troff -man` *file* ...

DESCRIPTION

These macros are used to format pages of this manual.

Except in `.LR` and `.RL` requests, any text argument denoted *t* in the request summary may be zero to six words. Quotes `"` ... `"` may be used to include blanks in a ‘word’. If *t* is empty, the special treatment is applied to the next text input line (the next line that doesn’t begin with dot). In this way, for example, `.I` may be used to italicize a line of more than 6 words, or `.SM` followed by `.B` to make small letters in ‘bold’ font.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

The fonts are

`R`	roman, the main font, preferred for diagnostics
`I`	italic, preferred for parameters, short names of commands names of manual pages, and naked function names
`B`	‘bold’, actually the constant width font CW, preferred for examples, file names, declarations, keywords, names of `struct` members, and literals (numbers are rarely literals)
`L`	also font CW. In *troff* `L=B`; in *nroff* arguments of the macros `.L`, `.LR`, and `.RL` are printed in quotes; preferred only where quotes really help (e.g. lower-case literals and punctuation).

Type font and size are reset to default values before each paragraph, and after processing font- or size-setting macros.

The `-man` macros admit equations and tables in the style of *eqn*(1) and *tbl*(1), but do not support arguments on `.EQ` and `.TS` macros.

These strings are predefined by `-man`:

`\*R`	‘®’, ‘(Reg)’ in *nroff.*
`\*S`	Change to default type size.

FILES

`/sys/lib/tmac/tmac.an`

SEE ALSO

troff(1), *man*(1)

REQUESTS

Request	Cause Break	If no Argument	Explanation
`.B` *t*	no	*t*=n.t.l.*	Text *t* is ‘bold’.
`.BI` *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and italic.
`.BR` *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and Roman.
`.DT`	no		Restore default tabs.
`.EE`	yes		End displayed example
`.EX`	yes		Begin displayed example
`.FR` *t*	no	*t*=n.t.l.	Join words of *t* alternating file name and Roman.
`.HP` *i*	yes	*i*=p.i.*	Set prevailing indent to *i*. Begin paragraph with hanging indent.
`.I` *t*	no	*t*=n.t.l.	Text *t* is italic.
`.IB` *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and bold.
`.IP` *x i*	yes	*x*=""	Same as `.TP` with tag *x*.
`.IR` *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and Roman.

`.L` *t*	no	*t*=n.t.l.	Text *t* is literal.
`.LP`	yes		Same as `.PP`.
`.LR` *t*	no		Join 2 words of *t* alternating literal and Roman.
`.PD` *d*	no	*d*=`.4v`	Interparagraph distance is *d.*
`.PP`	yes		Begin paragraph. Set prevailing indent to default.
`.RE`	yes		End of relative indent. Set prevailing indent to amount of starting `.RS`.
`.RF` *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and file name.
`.RI` *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and italic.
`.RL` *t*	no		Join 2 or 3 words of *t* alternating Roman and literal.
`.RS` *i*	yes	*i*=p.i.	Start relative indent, move left margin in distance *i.* Set prevailing indent to default for nested indents.
`.SH` *t*	yes	*t*=""	Subhead; reset paragraph distance.
`.SM` *t*	no	*t*=n.t.l.	Text *t* is small.
`.SS` *t*	no	*t*=""	Secondary subhead.
`.TF` *s*	yes		Prevailing indent is wide as string *s* in font `L`; paragraph distance is 0.
`.TH` *n c x*	yes		Begin page named *n* of chapter *c; x* is extra commentary, e.g. ‘local’, for page head. Set prevailing indent and tabs to default.
`.TP` *i*	yes	*i*=p.i.	Set prevailing indent to *i.* Restore default indent if *i*=0. Begin indented paragraph with hanging tag given by next text line. If tag doesn’t fit, place it on separate line.
`.1C`	yes		Equalize columns and return to 1-column output
`.2C`	yes		Start 2-column nofill output

* n.t.l. = next text line; p.i. = prevailing indent

BUGS

There’s no way to fool *troff* into handling literal double quote marks `"` in font-alternation macros, such as `.BI`.

There is no direct way to suppress column widows in 2-column output; the column lengths may be adjusted by inserting `.sp` requests before the closing `.1C`.

NAME

map − digitized map formats

DESCRIPTION

Files used by *map*(7) are a sequence of structures of the form:

```
struct {
      signed char patchlatitude;
      signed char patchlongitude;
      short n;
      union {
            struct {
                  short latitude;
                  short longitude;
            } point[n];
            struct {
                  short latitude;
                  short longitude;
                  struct {
                        signed char latdiff;
                        signed char londiff;
                  } point[-n];
            } highres;
      } segment;
};
```

where `short` stands for 16-bit integers and there is no padding within or between `structs`.

Fields `patchlatitude` and `patchlongitude` tell to what 10-degree by 10-degree patch of the earth's surface a segment belongs. Their values range from −9 to 8 and from −18 to 17, respectively, and indicate the coordinates of the southeast corner of the patch in units of 10 degrees.

Each segment of |`n`| points is connected; consecutive segments are not necessarily related. Latitude and longitude are measured in units of 0.0001 radian. If `n` is negative, then differences to the first and succeeding points are measured in units of 0.00001 radian. Latitude is counted positive to the north and longitude positive to the west.

The patches are ordered lexicographically by `patchlatitude` then `patchlongitude`. A printable index to the first segment of each patch in a file named *data* is kept in an associated file named *data*`.x`. Each line of an index file contains `patchlatitude, patchlongitude` and the byte position of the patch in the map file. Both the map file and the index file are ordered by patch latitude and longitude.

Shorts are stored in little-endian order, low byte first, regardless of computer architecture. To assure portability, *map* accesses them bytewise.

SEE ALSO

map(7)

NAME

mpictures − picture inclusion macros

SYNOPSIS

`troff -mpictures` [*options*] *file ...*

DESCRIPTION

Mpictures macros insert PostScript pictures into *troff*(1) documents. The macros are:

`.BP` *source height width position offset flags label*

Define a frame and place a picture in it. Null arguments, represented by `""`, are interpreted as defaults. The arguments are:

source	Name of a PostScript picture file, optionally suffixed with (*n*) to select page number *n* from the file (first page by default).
height	Vertical size of the frame, default `3.0i`.
width	Horizontal size of the frame, current line length by default.
position	`l` (default), `c`, or `r` to left-justify, center, or right-justify the frame.
offset	Move the frame horizontally from the original *position* by this amount, default `0i`.
flags	One or more of: `a`*d* Rotate the picture clockwise *d* degrees, default *d*=90. `o` Outline the picture with a box. `s` Freely scale both picture dimensions. `w` White out the area to be occupied by the picture. `l`,`r`,`t`,`b` Attach the picture to the left right, top, or bottom of the frame.
label	Place *label* at distance `1.5v` below the frame.

If there's room, `.BP` fills text around the frame. Everything destined for either side of the frame goes into a diversion to be retrieved when the accumulated text sweeps past the trap set by `.BP` or when the diversion is explicitly closed by `.EP`.

`.PI` *source height*`,` *width*`,` *yoffset*`,`*xoffset flags.*

This low-level macro, used by `.BP`, can help do more complex things. The two arguments not already described are:

xoffset	Offset the frame from the left margin by this amount, default `0i`.
yoffset	Offset the frame from the current baseline, measuring positive downward, default `0i`.

`.EP` End a picture started by `.BP`; `.EP` is usually called implicitly by a trap at frame bottom.

If a PostScript file lacks page-delimiting comments, the entire file is included. If no `%%BoundingBox` comment is present, the picture is assumed to fill an 8.5×11-inch page. Nothing prevents the picture from being placed off the page.

SEE ALSO

troff(1)

DIAGNOSTICS

A picture file that can't be read by the PostScript postprocessor is replaced by white space.

BUGS

A picture and associated text silently disappear if a diversion trap set by `.BP` isn't reached. Call `.EP` at the end of the document to retrieve it.

Macros in other packages may break the adjustments made to the line length and indent when text is being placed around a picture.

A missing or improper `%%BoundingBox` comment may cause the frame to be filled incorrectly.

NAME

mpm, mspe − macros for page makeup

SYNOPSIS

`troff -mpm` *file ...*

`troff -mspe` *file ...*

DESCRIPTION

These *troff*(1) macros, largely compatible with *ms*(6), make better pages. They silently invoke and provide information to a postprocessor that moves floating figures, avoids widows, and justifies pages vertically by stretching vertical spaces that result from `.PP`, `.LP`, `.IP`, `.QP`, `.SH`, `.NH`, `.DS`/`.DE`, `.EQ`/`.EN`, `.TS`/`.TE`, `.PS`/`.PE`, `.P1`/`.P2`, and `.QS`/`.QE`. The packages support different styles:

-mpm	generic
`-mspe`	*Software—Practice and Experience*

The following macros are different from or not part of `-ms`. Values denoted *n* have default value `1v`.

`.BP`	Begin a new page.
`.FL`	Flush: force out previous keeps.
`.FC`	Finish a two-column region and start a new one.
`.KF` *m*	Floating keep, with preferred center at vertical position *m*. Special values `top` (default) and `bottom` are permitted.
`.NE` *n*	Start new page if remaining vertical space on this page is less than *n* .
`.P1`	Begin a program display (Courier font).
`.P2`	End a program display.
`.P3`	Insert optional break point in program display.
`.SP` *n* `exactly`	
`.SP` *n*	Insert vertical space of height *n,* stretchable unless `exactly` is present.
`.Tm` *text*	Place page number and *text* on the standard error output.
`.X` *text*	Present *text* to the hidden page-makeup program as part of a device-dependent output sequence `x X` *text*. Equivalent to `\X'` *text*`'` .

Useful number registers:

`HM`	Header margin; default 1 inch.
`FM`	Footer margin; default 1 inch.
`FO`	Footer position; default 10 inches.
`%#`	Page number of current page.
`dP,dV`	Shrinkage of point size and vertical spacing for `.P1`, in points.

Useful strings:

`%e,%o`	Even and odd page title commands, as `.tl ''''`.

FILES

`/sys/lib/tmac/tmac.pm`
`/$cputype/bin/aux/pm`

SEE ALSO

ms(6), *troff*(1)
B. W. Kernighan and C. J. Van Wyk, ‘The −mpm Macro Package’, UNIX Research System Programmer’s Manual, Volume 2

BUGS

These features of `-ms` are missing:

- Document styles other than the default `.RP`.
- Space between front matter and first paragraph. Recover it with `.SP 2`.
- Separating rule above footnotes.
- Keeps assigned to a separate page.

Pages with more than two columns.

Troff option `-o` doesn't work with `-mpm` because only the postprocessor knows the page numbers.

NAME

ms − macros for formatting manuscripts

SYNOPSIS

`nroff −ms` [*options*] *file ...*
`troff −ms` [*options*] *file ...*

DESCRIPTION

This package of *nroff* and *troff*(1) macro definitions provides a canned formatting facility for technical papers in various formats.

The macro requests are defined below. Many *nroff* and *troff* requests are unsafe in conjunction with this package, but the following requests may be used with impunity after the first `.PP`: `.bp`, `.br`, `.sp`, `.ls`, `.na`.

Output of the *eqn*(1), *tbl*(1), and *pic*(1) preprocessors for equations, tables, pictures, and references is acceptable as input.

Diacritical marks may be applied to letters, as in these examples:

```
\*`e  \*`a  \*'e  \*^e  \*^o  \*:u  \*~n  \*,c  \*vc
è     à     é     ê     ô     ü     ñ     ç     č
```

FILES

`/sys/lib/tmac/tmac.s`

SEE ALSO

M. E. Lesk, ‘Typing Documents on the UNIX System: Using the −ms Macros with Troff and Nroff’, Unix Research System Programmer’s Manual, Volume 2
eqn(1), *troff*(1), *tbl*(1), *pic*(1)

REQUESTS

Request	Initial Value	Cause Break	Explanation
`.1C`	yes	yes	One column format on a new page.
`.2C`	no	yes	Two column format.
`.AB`	no	yes	Begin abstract.
`.AE`	-	yes	End abstract.
`.AI`	no	yes	Author’s institution follows. Suppressed in `.TM`.
`.AT`	no	yes	Print ‘Attached’ and turn off line filling.
`.AU` *x y*	no	yes	Author’s name follows. *x* is location and *y* is extension, ignored except in `TM`.
`.B` *x y*	no	no	Print *x* in boldface, append *y*; if no argument switch to boldface.
`.B1`	no	yes	Begin text to be enclosed in a box.
`.B2`	no	yes	End boxed text.
`.BI` *x y*	no	no	Print *x* in bold italic and append *y*; if no argument switch to bold italic.
`.BT`	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
`.BX` *x*	no	no	Print *x* in a box.
`.CW` *x y*	no	no	Constant width font (Courier) for *x*, append *y*; if no argument switch to CW.
`.CT`	no	yes	Print ‘Copies to’ and turn off line filling.
`.DA` *x*	nroff	no	‘Date line’ at bottom of page is *x*. Default is today.
`.DE`	-	yes	End displayed text. Implies `.KE`.
`.DS` *x*	no	yes	Start of displayed text, to appear verbatim line-by-line: `I` indented (default), `L` left-justified, `C` centered, `B` (block) centered with straight left margin. Implies `.KS`.
`.EG`	no	-	Print document in BTL format for ‘Engineer’s Notes.’ Must be first.
`.EN`	-	yes	Space after equation produced by *neqn* or *eqn*(1).
`.EQ` *x y*	-	yes	Display equation. Equation number is *y*. Optional *x* is `I`, `L`, `C` as in `.DS`.
`.FE`	-	yes	End footnote.
`.FP` *x*	-	no	Set font positions for a family, e.g., `.FP palatino`
`.FS`	no	no	Start footnote. The note will be moved to the bottom of the page.

`.HO`	-	no	‘Bell Laboratories, Holmdel, New Jersey 07733’.
`.I` *x y*	no	no	Italicize *x,* append *y*; if no argument switch to italic.
`.IH`	no	no	‘Bell Laboratories, Naperville, Illinois 60540’
`.IM`	no	no	Print document in BTL format for an internal memorandum. Must be first.
`.IP` *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
`.KE`	-	yes	End keep. Put kept text on next page if not enough room.
`.KF`	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
`.KS`	no	yes	Start keeping following text.
`.LG`	no	no	Make letters larger.
`.LP`	yes	yes	Start left-blocked paragraph.
`.LT`	no	yes	Start a letter; a non-empty first argument produces a full AT&T letterhead, a second argument is a room number, a third argument is a telephone number.
`.MF`	-	-	Print document in BTL format for ‘Memorandum for File.’ Must be first.
`.MH`	-	no	‘Bell Laboratories, Murray Hill, New Jersey 07974’.
`.MR`	-	-	Print document in BTL format for ‘Memorandum for Record.’ Must be first.
`.ND` *date*	troff	no	Use date supplied (if any) only in special BTL format positions; omit from page footer.
`.NH` *n*	-	yes	Same as `.SH`, with automatic section numbers like ‘1.2.3’; *n* is subsection level (default 1).
`.NL`	yes	no	Make letters normal size.
`.P1`	-	yes	Begin program display in Courier font.
`.P2`	-	yes	End program display.
`.PE`	-	yes	End picture; see *pic*(1).
`.PF`	-	yes	End picture; restore vertical position.
`.PP`	no	yes	Begin paragraph. First line indented.
`.PS` *h w*	-	yes	Start picture; height and width in inches.
`.PY`	-	no	‘Bell Laboratories, Piscataway, New Jersey 08854’
`.QE`	-	yes	End quoted material.
`.QP`	-	yes	Begin quoted paragraph (indent both margins).
`.QS`	-	yes	Begin quoted material (indent both margins).
`.R`	yes	no	Roman text follows.
`.RE`	-	yes	End relative indent level.
`.RP`	no	-	Cover sheet and first page for released paper. Must precede other requests.
`.RS`	-	yes	Start level of relative indentation from which subsequent indentation is measured.
`.SG` *x*	no	yes	Insert signature(s) of author(s), ignored except in `.TM` and `.LT`. *x* is the reference line (initials of author and typist).
`.SH`	-	yes	Section head follows, font automatically bold.
`.SM`	no	no	Make letters smaller.
`.TA` *x*...	5...	no	Set tabs in ens. Default is 5 10 15 ...
`.TE`	-	yes	End table; see *tbl*(1).
`.TH`	-	yes	End heading section of table.
`.TL`	no	yes	Title follows.
`.TM` *x*...	no	-	Print document in BTL technical memorandum format. Arguments are TM number, (quoted list of) case number(s), and file number. Must precede other requests.
`.TR` *x*	-	-	Print in BTL technical report format; report number is *x*. Must be first.
`.TS` *x*	-	yes	Begin table; if *x* is `H` table heading is repeated on new pages.
`.UL` *x*	-	no	Underline argument (even in troff).
`.UX` *y z*	-	no	‘*z*UNIX*y*’; first use gives registered trademark notice.
`.WH`	-	no	‘Bell Laboratories, Whippany, New Jersey 07981’.
`.]`	-	no	End reference.

NAME

namespace − name space description file

DESCRIPTION

Namespace files describe how to construct a name space from scratch, an operation normally performed by the *newns* suboroutine (see *auth*(2)) which is typically called by *init*(8). Each line specifies one name space operation. Spaces and tabs separate arguments to operations; no quotes or escapes are recognized. Blank lines and lines with `#` the first non-space character are ignored. Environment variables of the form `$`*name* are expanded within arguments, where *name* is a UTF string terminated by white space, a `/`, or a `$`.

The known operations and their arguments are:

`mount` [`-abct`] [`-s`*server*] *servename old* [*spec*]
Mount *servename* on *old*.

`bind` [`-abc`] *new old*
Bind *new* on *old*.

`import` [`-abct`] [`-s`*server*] *host* [*remotepath*] *mountpoint*
Import *remotepath* from machine *server* and attach it to *mountpoint.*

`cd` *dir* Change the working directory to *dir*.

The options for *bind, mount,* and *import* are interpreted as in *bind*(1) and *import*(4).

SEE ALSO

bind(1), *namespace*(4), *init*(8)

NAME

ndb − Network database

DESCRIPTION

The network database consists of the two files `/lib/ndb/local` and `/lib/ndb/global`. The files comprise multi-line entries made up of attribute/value pairs of the form *attr*=*value*. Each line starting without white space starts a new entry. Lines starting with `#` are comments.

Within entries pairs on the same line bind tighter than pairs on different lines.

The program `ndb/cs` (see *ndb*(8)) and the library routine `ipinfo` (see *ndb*(2)) perform searches for information relative to a particular host. `Ndb/cs` resolves meta-addresses of the form `$`*attribute* by returning the *value* from the *attribute*=*value* most closely related to the resolving host. The attribute-value pair comes from the entry for the system, its subnet, or its network with the system entry having precedence, subnet next, and network last.

A number of attributes are meaningful to programs and thus reserved. They are:

`sys`	system name
`dom`	Internet domain name
`ip`	Internet address
`ether`	Ethernet address
`dk`	Datakit address
`bootf`	file to download for initial bootstrap
`ipnet`	Internet network name
`ipmask`	Internet network mask
`ipgw`	Internet gateway
`auth`	authentication server to be used
`fs`	file server to be used
`tcp`	a TCP service name
`udp`	a UDP service name
`il`	an IL service name
`port`	a TCP, UDP, or IL port number
`restricted`	a TCP service that can be called only by ports numbered less that 1024
`proto`	a protocol supported by a host. The pair *proto*=*il* is needed by *cs* (see *ndb*(8)) in entries for hosts that support the IL protocol.
`9P`	parameters for the 9P file protocol, in particular whether the server authenticates (`9P=auth`).

EXAMPLES

An entry for the CPU server, spindle.

```
sys = spindle
      dom=spindle.research.att.com
      bootf=/mips/9powerboot
      ip=135.104.117.32 ether=080069020677
      dk=nj/astro/spindle
      proto=il
```

Entries for the network `mh-astro-net` and its subnets.

```
ipnet=mh-astro-net ip=135.104.0.0 ipmask=255.255.255.0
      fs=bootes.research.att.com
      ipgw=r70.research.att.com
      auth=p9auth.research.att.com
ipnet=unix-room ip=135.104.117.0
      ipgw=135.104.117.1
ipnet=third-floor ip=135.104.51.0
      ipgw=135.104.51.1
```

Mappings between TCP service names and port numbers.

```
tcp=sysmon     port=401
tcp=rexec      port=512   restricted
tcp=9fs        port=564
```

FILES

`/lib/ndb/local`
first database file searched

`/lib/ndb/global`
second database file searched

SEE ALSO

dial(2), *ndb*(2), *ndb*(8), *bootp*(8), *ipconfig*(8), *con*(1),

NAME

plot − graphics interface

DESCRIPTION

Files of this format are interpreted by *plot*(1) to draw graphics on the screen. A *plot* file is a UTF stream of instruction lines. Arguments are delimited by spaces, tabs, or commas. Numbers may be floating point. Punctuation marks (except `:`) , spaces, and tabs at the beginning of lines are ignored. Comments run from `:` to newline. Extra letters appended to a valid instruction are ignored. Thus `...line`, `line`, `li` all mean the same thing. Arguments are interpreted as follows:

1. If an instruction requires no arguments, the rest of the line is ignored.
2. If it requires a string argument, then all the line after the first field separator is passed as argument. Quote marks may be used to preserve leading blanks. Strings may include newlines represented as `\n`.
3. Between numeric arguments alphabetic characters and punctuation marks are ignored. Thus `line from 5 6 to 7 8` draws a line from (5, 6) to (7, 8).
4. Instructions with numeric arguments remain in effect until a new instruction is read. Such commands may spill over many lines. Thus the following sequence will draw a polygon with vertices (4.5, 6.77), (5.8, 5.6), (7.8, 4.55), and (10.0, 3.6).

```
move 4.5 6.77
vec 5.8, 5.6 7.8
4.55 10.0, 3.6 4.5, 6.77
```

The instructions are executed in order. The last designated point in a `line`, `move`, `rmove`, `vec`, `rvec`, `arc`, or `point` command becomes the 'current point' (*X,Y*) for the next command.

Open & Close

`o` *string*	Open plotting device. For *troff, string* specifies the size of the plot (default is `6i`).
`cl`	Close plotting device.

Basic Plotting Commands

`e`	Start another frame of output.
`m` *x y*	(move) Current point becomes *x y.*
`rm` *dx dy*	Current point becomes *X+dx Y+dy.*
`poi` *x y*	Plot the point *x y* and make it the current point.
`v` *x y*	Draw a vector from the current point to *x y.*
`rv` *dx dy*	Draw vector from current point to X + dx Y + dy
`li` *x1 y1 x2 y2*	Draw a line from *x1 y1* to *x2 y2.* Make the current point *x2 y2.*
`t` *string*	Place the *string* so that its first character is centered on the current point (default). If *string* begins with `\C` (`\R`), it is centered (right-adjusted) on the current point. A backslash at the beginning of the string may be escaped with another backslash.
`a` *x1 y1 x2 y2 xc yc r*	Draw a circular arc from *x1 y1* to *x2 y2* with center *xc yc* and radius *r.* If the radius is positive, the arc is drawn counterclockwise; negative, clockwise. The starting point is exact but the ending point is approximate.
`ci` *xc yc r*	Draw a circle centered at *xc yc* with radius *r.* If the range and frame parameters do not specify a square, the 'circle' will be elliptical.
`di` *xc yc r*	Draw a disc centered at *xc yc* with radius *r* using the filling color (see `cfill` below).
`bo` *x1 y1 x2 y2*	Draw a box with lower left corner at *x1 y1* and upper right corner at *x2 y2.*
`sb` *x1 y1 x2 y2*	Draw a solid box with lower left corner at *x1 y1* and upper right corner at *x2 y2* using the filling color (see `cfill` below).

`par` *x1 y1 x2 y2 xg yg*

Draw a parabola from *x1 y1* to *x2 y2* ‘guided’ by *xg yg.* The parabola passes through the midpoint of the line joining *xg yg* with the midpoint of the line joining *x1 y1* and *x2 y2* and is tangent to the lines from *xg yg* to the endpoints.

`pol { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Draw polygons with vertices *x1 y1 ... xn yn* and *X1 Y1 ... Xm Ym.* If only one polygon is specified, the inner brackets are not needed.

`fi { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Fill a polygon. The arguments are the same as those for `pol` except that the first vertex is automatically repeated to close each polygon. The polygons do not have to be connected. Enclosed polygons appear as holes.

`sp { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Draw a parabolic spline guided by *x1 y1 ... xn yn* with simple endpoints.

`fsp { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Draw a parabolic spline guided by *x1 y1 ... xn yn* with double first endpoint.

`lsp { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Draw a parabolic spline guided by *x1 y1 ... xn yn* with double last endpoint.

`dsp { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

Draw a parabolic spline guided by *x1 y1 ... xn yn* with double endpoints.

`csp { {`*x1 y1 ... xn yn*`} ... {`*X1 Y1 ... Xm Ym*`} }`

`in` *filename*

(include) Take commands from *filename.*

`de` *string* `{` *commands* `}`

Define *string* as *commands.*

`ca` *string scale*

Invoke commands defined as *string* applying *scale* to all coordinates.

Commands Controlling the Environment

`co` *string* Draw lines with color *string.* Possible colors: `black`, `red`, `green`, `blue`, `Tblack`, `Tred`, `Tgreen`, `Tblue`

`pe` *string* Use *string* as the style for drawing lines. The available pen styles are: `solid`, `dott`[ed], `short`, `long`, `dotd`[ashed]`,` `cdash`, `ddash`

`cf` *string* Color for filling (see `co`, above).

`ra` *x1 y1 x2 y2*

The data will fall between *x1 y1* and *x2 y2.* The plot will be magnified or reduced to fit the device as closely as possible.

Range settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1). 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 with nonsquare faces.

`fr` *px1 py1 px2 py2*

Plot the data in the fraction of the display specified by *px1 py1* for lower left corner and *px2 py2* for upper right corner. Thus `frame .5 0 1. .5` plots in the lower right quadrant of the display; `frame 0. 1. 1. 0.` uses the whole display but inverts the *y* coordinates.

`sa` Save the current environment, and move to a new one. The new environment inherits the old one. There are 7 levels.

`re` Restore previous environment.

SEE ALSO

plot(1), *graph*(1)

NAME

regexp − regular expression notation

DESCRIPTION

A *regular expression* specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. In many applications a delimiter character, commonly `/`, bounds a regular expression. In the following specification for regular expressions the word ‘character’ means any character (rune) but newline.

The syntax for a regular expression `e0` is

```
e3:  literal | charclass | '.' | '^' | '$' | '(' e0 ')'

e2:  e3
  |  e2 REP
REP: '*' | '+' | '?'

e1:  e2
  |  e1 e2

e0:  e1
  |  e0 '|' e1
```

A `literal` is any non-metacharacter or a metacharacter (one of `.*+?[]()|\^$`) or the delimiter preceded by `\`.

A `charclass` is a nonempty string *s* bracketed `[` *s* `]` (or `[^`*s* `]`); it matches any character in (or not in) *s.* A negated character class never matches newline. A substring *a*−*b*, with *a* and *b* in ascending order, stands for the inclusive range of characters between *a* and *b*. In *s,* the metacharacters `-`, `]`, an initial `^`, and the regular expression delimiter must be preceded by a `\`; other metacharacters have no special meaning and may appear unescaped.

A `.` matches any character.

A `^` matches the beginning of a line; `$` matches the end of the line.

The `REP` operators match zero or more (`*`), one or more (`+`), zero or one (`?`), instances respectively of the preceding regular expression `e2`.

A concatenated regular expression, `e1 e2`, matches a match to `e1` followed by a match to `e2`.

An alternative regular expression, `e0 | e1`, matches either a match to `e0` or a match to `e1`.

A match to any part of a regular expression extends as far as possible without preventing a match to the remainder of the regular expression.

SEE ALSO

awk(1), *ed*(1), *sam*(1), *sed*(1), *regexp*(2)

NAME

users − file server user list format

DESCRIPTION

The permanent file servers each maintain a private list of users and groups, in `/adm/users` by convention. Each line in the file has the format

num:*name*:*leader*:*members*

where *num* is a decimal integer, *name* and *leader* are printable strings excluding the characters `?`, `=`, `+`, `−`, `/`, and `:`, and *members* is a comma-separated list of such strings. Such a line defines a user and a group with the given *name*; the group has a group leader given by *leader* and group members given by the user names in *members*. The *leader* field may be empty, in which case any group member is a group leader. The *members* field may be empty.

Lines beginning with `#` are ignored.

The *num* in a line is a number used internally by a file server; there should be no duplicate *num*s in the file. A negative *num* is special: a user with a negative *num* cannot attach to the file server. The file `/adm/users` itself is owned by user *adm*, having a negative *num*, so it can only be changed via console commands.

SEE ALSO

intro(5), *stat*(5)

NAME

UTF, Unicode, ASCII, rune − character set and format

DESCRIPTION

The Plan 9 character set and representation are based on Unicode and on a proposed X-Open multibyte FSS-UCS-TF (File System Safe Universal Character Set Transformation Format) encoding. Unicode represents its characters in 16 bits; FSS-UCS-TF, or just UTF, represent such values in an 8-bit byte stream.

In Plan 9, a *rune* is a 16-bit quantity representing a Unicode character. Internally, programs may store characters as runes. However, any external manifestation of textual information, in files or at the interface between programs, uses a machine-independent, byte-stream encoding called UTF.

UTF is designed so the 7-bit ASCII set (values hexadecimal 00 to 7F), appear only as themselves in the encoding. Runes with values above 7F appear as sequences of two or more bytes with values only from 80 to FF.

The UTF encoding of Unicode is backward compatible with ASCII: programs presented only with ASCII work on Plan 9 even if not written to deal with UTF, as do programs that deal with uninterpreted byte streams. However, programs that perform semantic processing on ASCII graphic characters must convert from UTF to runes in order to work properly with non-ASCII input. See *rune*(2).

Letting numbers be binary, a rune x is converted to a multibyte UTF sequence as follows:

01. x in [00000000.0bbbbbbb] → 0bbbbbbb
10. x in [00000bbb.bbbbbbbb] → 110bbbbb, 10bbbbbb
11. x in [bbbbbbbb.bbbbbbbb] → 1110bbbb, 10bbbbbb, 10bbbbbb

Conversion 01 provides a one-byte sequence that spans the ASCII character set in a compatible way. Conversions 10 and 11 represent higher-valued characters as sequences of two or three bytes with the high bit set. Plan 9 does not support the 4, 5, and 6 byte sequences proposed by X-Open. When there are multiple ways to encode a value, for example rune 0, the shortest encoding is used.

In the inverse mapping, any sequence except those described above is incorrect and is converted to rune 0080.

FILES

`/lib/unicode`

table of characters and descriptions, suitable for *look*(1).

SEE ALSO

ascii(1), *tcs*(1), *rune*(2), *keyboard*(6), *The Unicode Standard*.

NAME

intro − introduction to databases

DESCRIPTION

This manual section describes databases available on Plan 9 and the commands that access them. Some of them involve proprietary data that is not distributed outside Bell Laboratories.

NAME

ahd − American Heritage Dictionary

SYNOPSIS

`ahd` [`−p`] [*word...*]

DESCRIPTION

Given one or more *word* arguments, *ahd* prints the matching dictionary entry or entries. Otherwise, words to be looked up are taken from the standard input, one per line. Under the `−p` option, entries match if they have the specified word as a prefix.

FILES

`/lib/ahd/*`	dictionary files.
`/$cputype/bin/aux/ahddisplay`	display program.

NAME

astro − print astronomical information

SYNOPSIS

`astro [ -dlepsatokm ] [ -c`*n*` ]`

DESCRIPTION

Astro reports upcoming celestial events, by default for 24 hours starting now. The options are:

`d`	Read the starting date. A prompt gives the input format.
`l`	Read the north latitude, west longitude, and elevation of the observation point A prompt gives the input format. If `l` is missing, the initial position is read from the file `/lib/sky/here`.
`c`	Report for *n* (default 1) successive days.
`e`	Report fractional overlap during eclipses.
`p`	Print the positions of objects at the given time rather than searching for interesting conjunctions. For each, the name is followed by the right ascension (hours, minutes, seconds), declination (degrees, minutes, seconds), azimuth (degrees), elevation (degrees), and semidiameter (arc minutes). For the sun and moon, the magnitude is also printed.
`s`	Print output in English words suitable for speech synthesizers.
`a`	Include a list of artificial earth satellites for interesting events. (There are no orbital elements for the satellites, so this option is not usable.)
`t`	Read ΔT from standard input. ΔT is the difference between ephemeris and universal time (seconds) due to the slowing of the earth's rotation. ΔT is normally calculated from an empirical formula. This option is needed only for very accurate timing of occultations, eclipses, etc.
`o`	Search for stellar occultations.
`k`	Print times in local time ('kitchen clock') as described in the `timezone` environment variable.
`m`	Includes a single comet in the list of objects. This is modified (in the source) to refer to an approaching comet but in steady state usually refers to the last interesting comet (currently Levy, 1990c).

FILES

`/lib/sky/estartab`	ecliptic star data
`/lib/sky/here`	default latitude (N), longitude (W), and elevation (meters)

SEE ALSO

scat(7)

BUGS

The `k` option reverts to GMT outside of 1970-2036.

NAME

chdb − chess database browser

SYNOPSIS

`chdb` [*file* ...]

DESCRIPTION

Chdb reads the given *files* of chess games (`hist` by default) and accepts commands to search, play through, and display the games in these files. If *x* is the name of the file, it is looked for under the names *x*, *x*`.m.out`, and `/lib/chess/`*x*`.m.out`.

After reading the files, *chdb* displays a chess board, a text window with a command line, and vertical and horizontal scroll bars along the edges of the board. The text window holds 6 lines of information about the game and position and, at the bottom, a command line in which to type. The displayed text contains chess symbols—use a suitable font.

The vertical scroll bar scans through the games to select a game; the horizontal bar then scans through the moves of the selected game. Both scroll bars use button 1 to scan backwards, button 3 to scan forwards, and button 2 to jump to an absolute position. With buttons 1 and 3, the scroll bars are calibrated in units of 1 through 8 corresponding to the ranks and files of the board. The units measure games on the vertical scroll bar; ply (half-moves) on the horizontal scroll bar. For example, clicking button 3 on the horizontal scroll bar under the `d` file steps through the game two full moves.

Moves may be made on the chess board by pointing with button 1. There are two methods to point at moves. For the first method, point at the piece to be moved, press button 1, point to the place to move that piece, and then release button 1. In the second method, point at the place where a piece is to move and press button 1. The smallest/least-central piece that can move there is highlighted. Releasing button 1 without moving the mouse will make the highlighted move. Moving the mouse to the desired piece and releasing the button will move the selected piece.

Typed lines of text are echoed in the command line and executed. The available commands are:

`f` *n n* Set the format for display of moves in the text window. The first number is the verbosity, with 0 minimal. The second number is 0 for algebraic, 1 for English, and 2 for figurine (default).

`g` *n* Go to the game with ordinal number *n* in the input files. If *n* is prefixed with a + or `−` , it is interpreted as a relative position in the current set of games (see below). *N* defaults to `+1` .

`p` *n* Go to whole move *n* in the current game. If *n* is prefixed with a + or `−` , it is interpreted as an offset in *ply* from the current position. *N* defaults to `+1` .

`k` *n* Mark the current set of games with tag *n* (see patterns, below).

`w` *type file*
Write the current set of games to *file*. *Type* is either `a` (write the games as text) or `m` (write the games in binary format suitable for *chdb*).

Patterns select subsets of the games. A pattern is one of the following, in decreasing precedence order. Parentheses can be used for grouping.

`.` The current game.

`*` All games originally read.

`'` *n* The games previously marked with a `k` command with the same *n*.

`[]`

`[`*number*`]`
All games in `*` that contain the positions that can be reached in the specified number of plies from the current position. A missing number is the same as zero, meaning just the current position. Positions are matched with black/white transpositions.

/ regular expression / fields

This pattern matches the regular expression against the various text windows. *Fields* is a list of characters from the set `abdefoprw`. `A` is for all, `b` for black, `d` for date, `e` for event, `f` for file, `o` for opening, `p` for person (white and black), `r` for result, and `w` for white. If multiple fields are given, the expression is matched on the union of the specified fields. If no field is given, `p` is assumed.

! *pattern*

The set subtraction of `*` and the given pattern.

pattern − pattern

The set subtraction of the given patterns.

pattern + pattern

pattern | pattern

The set union of the given patterns.

pattern pattern

pattern `&` *pattern*

The set intersection of the given patterns.

+*pattern*

−*pattern*

`&`*pattern*

| *pattern*

These patterns have the current set of games as an implied first operand.

EXAMPLE

Select games that Botvinnik lost:

```
(/Botv/w/0-1/r) | (/Botv/b/1-0/r)
```

FILES

`/lib/chess` directory of databases.

SEE ALSO

regexp(6).

BUGS

The browser is only a prototype.

Most of the databases are protected by copyright and not distributed.

NAME

dict − dictionary browser

SYNOPSIS

`dict` [`-k`] [`-d` *dictname*] [*pattern*]

DESCRIPTION

Dict is a dictionary browser. If a *pattern* is given on the command line, *dict* prints all matching entries; otherwise it repeatedly accepts and executes commands. The options are

`-d` *dictname*	Use the given dictionary. The default is *oed*, the second edition of the Oxford English Dictionary. Available dictionaries printed with `-d?`
`-k`	Print a pronunciation key.

Patterns are regular expressions (see *regexp*(6)), with an implicit leading `^` and trailing `$`. Patterns are matched against an index of headwords and variants, to form a 'match set'. By default, both patterns and the index are folded: uppercase characters are mapped into their lowercase equivalents, and Latin-1 accented characters are mapped into their non-accented equivalents. In interactive mode, there is always a 'current match set' and a 'current entry' within the match set. Commands can change either or both, as well as print the entries or information about them.

Commands have an address followed by a command letter. Addresses have the form:

/ *re* /	Set the match set to all entries matching the regular expression *re*, sorted in dictionary order. Set the current entry to the first of the match set.
! *re* !	Like / *re* / but use exact matching, i.e., without case and accent folding.
/ *re* / . *n*	Like / *re* / but set the current entry to the *n*th of the match set.
! *re* ! . *n*	Like / *re* / . *n* but without folding.
. *n*	Just change the current entry to the *n*th of the current match set.
n	An integer *n* is an absolute byte offset into the raw dictionary. (See the `A` command, below.)
addr+	After setting the match set and current entry according to *addr*, change the match set and current entry to be the next entry in the dictionary (not necessarily in the match set) after the current entry.
addr−	Like *addr*+ but go to previous dictionary entry.

The command letters come in pairs: a lowercase and the corresponding uppercase letter. The lowercase version prints something about the current entry only, and advances the current entry to the next in the match set (wrapping around to the beginning after the last). The uppercase version prints something about all of the match set and resets the current entry to the beginning of the set.

`p,P`	Print the whole entry.
`h,H`	Print only the headword(s) of the entry.
`a,A`	Print the dictionary byte offset of the entry.
`r,R`	Print the whole entry in raw format (without translating special characters, etc.).

If no command letter is given, the last-typed command letter is used.

FILES

`/lib/oed/oed2`
`/lib/oed/oed2index`

SEE ALSO

regexp(6)

BUGS

A unicode font (e.g. `/lib/font/bit/pelm/unicode.9.font`) should be used for best results.

If the regular expression pattern doesn't begin with a few literal characters, matching takes a long time.

NAME

map, mapdemo − draw maps on various projections

SYNOPSIS

`map` *projection* [*option ...*]

`mapdemo`

DESCRIPTION

Map prepares on the standard output a map suitable for display by any plotting filter described in *plot*(1). A menu of projections is produced in response to an unknown *projection*. *Mapdemo* is a short course in mapping.

The default data for *map* are world shorelines. Option `-f` accesses more detailed data classified by feature.

`-f` [*feature ...*]

Features are ranked 1 (default) to 4 from major to minor Higher-numbered ranks include all lower-numbered ones. Features are

`shore[1-4]`	seacoasts, lakes, and islands; option `-f` always shows `shore1`
`ilake[1-2]`	intermittent lakes
`river[1-4]`	rivers
`iriver[1-3]`	intermittent rivers
`canal[1-3]`	`3`=irrigation canals
`glacier`	
`iceshelf[12]`	
`reef`	
`saltpan[12]`	
`country[1-3]`	`2`=disputed boundaries, `3`=indefinite boundaries
`state`	states and provinces (US and Canada only)

In other options coordinates are in degrees, with north latitude and west longitude counted as positive.

`-l` *S N E W*

Set the southern and northern latitude and the eastern and western longitude limits. Missing arguments are filled out from the list −90, 90, −180, 180, or lesser limits suitable to the projection at hand.

`-k` *S N E W*

Set the scale as if for a map with limits `-l` *S N E W*. Do not consider any `-l` or `-w` option in setting scale.

`-o` *lat lon rot*

Orient the map in a nonstandard position. Imagine a transparent gridded sphere around the globe. Turn the overlay about the North Pole so that the Prime Meridian (longitude 0) of the overlay coincides with meridian *lon* on the globe. Then tilt the North Pole of the overlay along its Prime Meridian to latitude *lat* on the globe. Finally again turn the overlay about its ‘North Pole’ so that its Prime Meridian coincides with the previous position of meridian *rot*. Project the map in the standard form appropriate to the overlay, but presenting information from the underlying globe. Missing arguments are filled out from the list 90, 0, 0. In the absence of `-o,` the orientation is 90, 0, *m,* where *m* is the middle of the longitude range.

`-w` *S N E W*

Window the map by the specified latitudes and longitudes in the tilted, rotated coordinate system. Missing arguments are filled out from the list −90, 90, −180, 180. (It is wise to give an encompassing `-l` option with `-w`. Otherwise for small windows computing time varies inversely with area!)

`-d` *n* For speed, plot only every *n*th one.

`-r` Reverse left and right, (good for star charts and inside-out views).

`-s` Save the screen, don’t erase before drawing. Output made under `-s` must be appended to output of another *map* command.

`-g` *dlat dlon res*

Grid spacings are *dlat, dlon.* Zero spacing means no grid. Missing *dlat* is taken to be zero. Missing *dlon* is taken the same as *dlat*. Grid lines are drawn to a resolution of *res* (2° or less by default). In the absence of `-g,` grid spacing is 10°.

`-p` *lat lon extent*

Position the point *lat, lon* at the center of the plotting area. Scale the map so that the height (and width) of the nominal plotting area is *extent* times the size of one degree of latitude at the center. By default maps are scaled and positioned to fit within the plotting area. An *extent* overrides option `-k`.

`-c` *x y rot*

After all other positioning and scaling operations have been performed, rotate the image *rot* degrees counterclockwise about the center and move the center to position *x, y,* where the nominal plotting area is $-1 \le x \le 1$, $-1 \le y \le 1$. Missing arguments are taken to be 0.

`-m` [*file* ...]

Use map data from named files. If no files are named, omit map data. Names that do not exist as pathnames are looked up in a standard directory, which contains, in addition to the data for `-f`,

`world`	World Data Bank I (default)
`states`	US map from Census Bureau
`counties`	US map from Census Bureau

The environment variables `MAP` and `MAPDIR` change the default map and default directory.

`-b` [*lat0 lon0 lat1 lon1*...]

Suppress the drawing of the normal boundary (defined by options `-l` and `-w`). Coordinates, if present, define the vertices of a polygon to which the map is clipped. If only two vertices are given, they are taken to be the diagonal of a rectangle. To draw the polygon, give its vertices as a `-u` track.

`-t` *file* ...

The *files* contain lists of points, given as latitude-longitude pairs in degrees. If the first file is named `-`, the standard input is taken instead. The points of each list are plotted as connected ‘tracks’.

Points in a track file may be followed by label strings. A label breaks the track. A label may be prefixed by `"`, `:`, or `!` and is terminated by a newline. An unprefixed string or a string prefixed with `"` is displayed at the designated point. The first word of a `:` or `!` string names a special symbol (see option `-y`). An optional numerical second word is a scale factor for the size of the symbol, 1 by default. A `:` symbol is aligned with its top to the north; a `!` symbol is aligned vertically on the page.

`-u` *file* ...

Same as `-t`, except the tracks are unbroken lines. (`-t` tracks appear as dot-dashed lines if the plotting filter supports them.)

`-y` *file* The *file* contains *plot*(6)-style data for `:` or `!` labels in `-t` or `-u` files. Each symbol is defined by a comment `:`*name* then a sequence of `m` and `v` commands. Coordinates (0,0) fall on the plotting point. Default scaling is as if the nominal plotting range were `ra -1 -1 1 1`; `ra` commands in *file* change the scaling.

Projections

Equatorial projections centered on the Prime Meridian (longitude 0). Parallels are straight horizontal lines.

`mercator`	equally spaced straight meridians, conformal, straight compass courses
`sinusoidal`	equally spaced parallels, equal-area, same as `bonne 0`.
`cylequalarea` *lat0*	equally spaced straight meridians, equal-area, true scale on *lat0*
`cylindrical`	central projection on tangent cylinder
`rectangular` *lat0*	equally spaced parallels, equally spaced straight meridians, true scale on *lat0*
`gall` *lat0*	parallels spaced stereographically on prime meridian, equally spaced straight meridians, true scale on *lat0*
`mollweide`	(homalographic) equal-area, hemisphere is a circle

Azimuthal projections centered on the North Pole. Parallels are concentric circles. Meridians are equally spaced radial lines.

`azequidistant`	equally spaced parallels, true distances from pole
`azequalarea`	equal-area
`gnomonic`	central projection on tangent plane, straight great circles
`perspective` *dist*	viewed along earth's axis *dist* earth radii from center of earth
`orthographic`	viewed from infinity
`stereographic`	conformal, projected from opposite pole
`laue`	*radius* = tan(2×*colatitude*), used in xray crystallography
`fisheye` *r*	*radius* = log(*colatitude*/*r*): *New Yorker* map from viewing pedestal of radius *r* degrees

Polar conic projections symmetric about the Prime Meridian. Parallels are segments of concentric circles. Except in the Bonne projection, meridians are equally spaced radial lines orthogonal to the parallels.

`conic` *lat0*	central projection on cone tangent at *lat0*
`simpleconic` *lat0 lat1*	equally spaced parallels, true scale on *lat0* and *lat1*
`lambert` *lat0 lat1*	conformal, true scale on *lat0* and *lat1*
`albers` *lat0 lat1*	equal-area, true scale on *lat0* and *lat1*
`bonne` *lat0*	equally spaced parallels, equal-area, parallel *lat0* developed from tangent cone

Projections with bilateral symmetry about the Prime Meridian and the equator.

`polyconic`	parallels developed from tangent cones, equally spaced along Prime Meridian
`aitoff`	equal-area projection of globe onto 2-to-1 ellipse, based on *azequalarea*
`lagrange`	conformal, maps whole sphere into a circle
`bicentric` *lon0*	points plotted at true azimuth from two centers on the equator at longitudes ±*lon0,* great circles are straight lines (a stretched *gnomonic*)
`elliptic` *lon0*	points plotted at true distance from two centers on the equator at longitudes ±*lon0*
`globular`	hemisphere is circle, circular arc meridians equally spaced on equator, circular arc parallels equally spaced on 0- and 90-degree meridians
`vandergrinten`	sphere is circle, meridians as in *globular,* circular arc parallels resemble *mercator*

Doubly periodic conformal projections.

`guyou`	W and E hemispheres are square
`square`	world is square with Poles at diagonally opposite corners
`tetra`	map on tetrahedron with edge tangent to Prime Meridian at S Pole, unfolded into equilateral triangle
`hex`	world is hexagon centered on N Pole, N and S hemispheres are equilateral triangles

Miscellaneous projections.

`harrison` *dist angle*	oblique perspective from above the North Pole, *dist* earth radii from center of earth, looking along the Date Line *angle* degrees off vertical

`trapezoidal` *lat0 lat1*	equally spaced parallels, straight meridians equally spaced along parallels, true scale at *lat0* and *lat1* on Prime Meridian

Retroazimuthal projections. At every point the angle between vertical and a straight line to 'Mecca', latitude *lat0* on the prime meridian, is the true bearing of Mecca.

`mecca` *lat0*	equally spaced vertical meridians
`homing` *lat0*	distances to Mecca are true

Maps based on the spheroid. Of geodetic quality, these projections do not make sense for tilted orientations. For descriptions, see corresponding maps above.

`sp_mercator`
`sp_albers` *lat0 lat1*

EXAMPLES

```
map perspective 1.025 -o 40.75 74
```

A view looking down on New York from 100 miles (0.025 of the 4000-mile earth radius) up. The job can be done faster by limiting the map so as not to 'plot' the invisible part of the world: `map perspective 1.025 -o 40.75 74 -l 20 60 30 100`. A circular border can be forced by adding option `-w 77.33`. (Latitude 77.33° falls just inside a polar cap of opening angle arccos(1.025) = 12.6804°.)

```
map mercator -o 49.25 -106 180
```

An 'equatorial' map of the earth centered on New York. The pole of the map is placed 90° away (40.75+49.25=90) on the other side of the earth. A 180° twist around the pole of the map arranges that the 'Prime Meridian' of the map runs from the pole of the map over the North Pole to New York instead of down the back side of the earth. The same effect can be had from `map mercator -o 130.75 74`

```
map albers 28 45 -l 20 50 60 130 -m states
```

A customary curved-latitude map of the United States.

```
map harrison 2 30 -l -90 90 120 240 -o 90 0 0
```

A fan view covering 60° on either side of the Date Line, as seen from one earth radius above the North Pole gazing at the earth's limb, which is 30° off vertical. The `-o` option overrides the default `-o 90 0 180`, which would rotate the scene to behind the observer.

FILES

`/lib/map/[1-4]??`	World Data Bank II, for `-f`
`/lib/map/*`	maps for `-m`
`/lib/map/*.x`	map indexes
`/bin/aux/mapd`	Map driver program

SEE ALSO

map(6), *plot*(1), *tiger*(7)

DIAGNOSTICS

'Map seems to be empty'—a coarse survey found zero extent within the `-l` and `-w` bounds; for maps of limited extent the grid resolution, *res,* or the limits may have to be refined.

BUGS

Windows (option `-w`) cannot cross the Date Line.
No borders appear along edges arising from visibility limits.
Segments that cross a border are dropped, not clipped.
Excessively large scale or `-d` setting may cause long line segments to be dropped.
Map tries to draw grid lines dotted and `-t` tracks dot-dashed. As very few plotting filters properly support curved textured lines, these lines are likely to appear solid.
The west-longitude-positive convention betrays Yankee chauvinism.

NAME

oed − Oxford English Dictionary

SYNOPSIS

`oed` [*options*] [`-[iI]` *index*] *word...*
`oed -b`*addr* [*options...*] [*sections...*]

DESCRIPTION

Given one or more *word* arguments, `oed` prints the matching main entry or entries from the first edition of the OED with Supplements. Flag options for this case are:

`-p`	Match all entries having *word* as a prefix.
`-I`*index*	Print sections of the dictionary in which, according to the *index*, the *word* appears.
`-i`*index*	Like `-I`, but only head words are printed, along with a command that will print the rest.

Available indexes are:

`le`	*Word* is a lexical entry (the default).
`et`	*Word* appears in an etymology article.
`se`	Sense: *word* is used in a definition.
`la`	*Word* is used as a label.
`qd`	*Word* appears in the date of a quoted work.
`qa`	*Word* is the author of a quoted work.
`qw`	*Word* is the title of a quoted work.
`qt`	*Word* appears in the text of a quoted work.

Except in the case of `le`, the program prints only the sections of an entry relevant to the *word*, e.g., indexing through `qt` produces the head word, a sense article, and the quotation.

The second form of `oed` uses file block addressing, primarily for the use of `oed -i...`. Options are:

`-a`	Print starting addresses and tags for each section.
`-b`*addr*	Print the dictionary starting with block *addr*.
`-b`*a*`.`*d*	Print the dictionary starting with the *d*'th definition (counting from zero) in block *a*. Printing stops at the end of this definition, or when the block specified with `-e` is reached.
`-e`*addr*	Stop printing when block *addr* is reached.

Specific *sections* or parts thereof may be selected within a definition, as follows:

n	Section number *n* (decimal).
n.*ttt*	Parts of section *n* (decimal) having tag *ttt* (hex).
n.*ttt*.*mmm*	Parts of section *n* (decimal) having tag *ttt* (hex), but only tag bits *mmm* (hex) are significant in the comparison.

Options applicable to both forms are:

`-k`	Print the pronunciation key and exit.
`-r`	Print the raw text from the dictionary, instead of a more readable form.

EXAMPLES

```
oed poot
```

Look up the word `poot`.

```
oed -i qt poot
```

Show words used with `poot` in quoted text.

```
oed -i la spiritualism
```

Show words cited as terms of art in spiritualism.

FILES

`/lib/oed/rcd0`	CDROM image.

BUGS

The `qa` and `qw` indexes make heavy use of unpredictable abbreviations. Entries for dates in the `qd` index include references that are *ante*, *post*, and *circa*; the program does not distinguish these.

Tabular and other typographically complicated material is missing from the database, and flagged by `{...}`.

A cross-reference that is not a main entry may be missing from the `le` index (e.g., *kinesthesia* refers to *kinæsthesia*, but the definition is found under *kinæsthesis*). The `-p` flag with a long prefix of the desired word is often successful.

NAME

scat − sky catalogue

SYNOPSIS

`scat`

DESCRIPTION

Scat looks up items in catalogues of objects outside the solar system and implements database-like manipulations on sets of such objects.

Items are read, one per line, from the standard input and looked up in the catalogs. The result of the lookup becomes the set of objects available to the database commands. After each lookup or command, if more than two objects are in the set, *scat* prints how many objects are in the set; otherwise it prints the objects' descriptions or cross-index listings (suitable for input to *scat*). An item is in one of the following formats:

`ngc1234`
: Number 1234 in the Revised New General Catalogue of Nonstellar Objects. The output identifies the type (`eg`=galaxy, `pn`=planetary nebula, `gc`=globular cluster, `oc`=open cluster, `dn`=diffuse nebula or `nc`=nebular cluster), possibly contained within the Large Magellanic Cloud (`in lmc`) or Small Magellanic Cloud (`in smc`), its position in 2000.0 coordinates and galactic coordinates, and a brief description.

`sao12345`
: Number 12345 in the Smithsonian Astrophysical Star Catalogue. Output identifies the visual and photographic magnitudes, 2000.0 coordinates, proper motion, spectral type, multiplicity and variability class, and HD number.

`m4`
: Catalog number 4 in Messier's catalog. The output is the NGC number.

`planetarynebula`
: The set of NGC objects of the specified type. The type may be a two-letter NGC code or a full name, as above, with no blank.

`"α umi"`
: Star names are provided in double quotes. Known names are the Greek letter designations, proper names such as Betelgeuse, and bright variable stars. Greek letters may be spelled out, e.g. `alpha`. Constellation names must be the three-letter abbreviations. The output is the SAO number. For non-Greek names, SAO numbers and names are listed for all stars with names for which the given name is a prefix.

`12h34m -16`
: Coordinates in the sky are translated to the nearest 'patch', approximately one square degree of sky. The output is the coordinates identifying the patch, the constellations touching the patch, and the NGC and SAO objects in the patch. The program prints sky positions in several formats corresponding to different precisions; any output format is understood as input.

`umi`
: All the patches in the named constellation.

The commands are:

`add` *item*
: Add the named item to the set.

`keep` *class ...*
: Flatten the set and cull it, keeping only the specified classes. The classes may be specific NGC types, all stars (`sao`), all NGC objects (`ngc`), all M objects (`m`), or a specified brightness range. Brightness ranges are specified by a leading > or < followed by a magnitude. Remember that brighter objects have lesser magnitudes.

`drop` *class ...*
: Like *keep,* but keeps only the objects not in the specified classes.

`flat` Some items such as patches represents sets of items. *Flat* flattens the set so *scat* holds all the information available for the objects in the set.

`print` Print the contents of the set. If the information seems meagre, try flattening the set.

`expand` *n*
Flatten the set, expand the area of the sky covered by the set to be *n* degrees wider, and collect all the objects in that area. If *n* is zero, *expand* collects all objects in the patches that cover the current set.

`plot` *option*
Expand and plot the set on the screen. The only option is `nogrid` to suppress the lines of declination and right ascension. Symbols for NGC objects are as in Sky Atlas 2000.0.

EXAMPLES

Plot the NGC objects and naked-eye stars in Orion.

```
ori
keep ngc <6
plot nogrid
```

Draw a map of the Pleiades.

```
"alcyone"
expand 1
plot
```

FILES

`/lib/sky/*.scat`

SEE ALSO

astro(7)

`/lib/sky/constelnames` for the three-letter abbreviations of the constellation names.

The data was provided by the Astronomical Data Center at the NASA Goddard Space Flight Center.

NAME

showimage − bitmap displayer, colormap changer

SYNOPSIS

`showimage` [*option*] *file ...*

DESCRIPTION

Showimage displays the bitmap contained in *file* in the top left corner of the current window. It goes to the next file when any character is typed, exiting when there are no more files. The options are:

`−c` Load the standard Plan 9 colormap, if this is an 8-bit display. The standard colormap takes the value of a pixel byte and uses the top 3 bits for red darkness, the next 3 bits for green darkness, and the final 2 for blue darkness. So 0 is white and 255 is black. Exceptions: pixels 85 and 170 are intermediate grey values, so that 2-bit-per-pixel grey scale images look right.

`−g` Load the colormap with a linear grey scale, from 0 (white) to max (black).

`−r` Load the colormap with a reverse linear grey scale, from 0 (black) to max (white).

`−m`*mapfile*
Load the colormap from the file *mapfile*. See *rgbpix*(2) for the format.

`−d` Dump the current colormap to standard output in the format of *rgbpix*(2).

FILES

`/lib/image` Some sample pictures.

SEE ALSO

graphics(2), *rgbpix*(2)

NAME

tiger − United States street map database

SYNOPSIS

`tiger` [county[,state] ...]

DESCRIPTION

Tiger displays *counties* from the U.S. Census Tiger Database, a street map of the entire U.S. and affiliates such as Puerto Rico, Samoa, the Marshall Islands, etc.

Keyboard commands

`s#` sets the scale of the map to the number. There are 4 scales per power of 10 and scale 5 is 5 nautical miles across the screen. Scale 16 (2800 nmi.) is a little larger than the width of the U.S. Scale 2 is about one statute mile. From scale 9 to 18, only primary and secondary roads are displayed. From scales 0 to 8, all roads, railroads and water features are displayed.

/*regexp*

All features labeled by text matching the regular expression are highlighted, even for features not displayed at the current scale.

`r`*county,state*

adds the map of the named *county* to the current display list. The center of the display is set to the center of the resulting display list. The scale is set to 9. A *county* can be the name of a county if it is unique (`sandiego`) or have a comma and state abbreviation if the county is not unique (`union,nj`). The county can also be a six digit code of the form SSCCCC taken from the database. See the file `/lib/tiger/codes1` for counties and their codes.

`e`*county,state*

is the same as *r* except that the previous display list is discarded before the new county is read.

`q` Exit *tiger.*

Mouse button commands

Button 1

displays the text of the *line* nearest the cursor. Only displayed lines are examined.

Button 2

displays the text of the *area* nearest the cursor. All areas are examined. Areas (as opposed to lines) include oceans, lakes, wide rivers, large cemeteries, some airports, counties, cities, and a lot more.

Button 3

centers the display on the cursor position and redraws the map.

FILES

`/lib/tiger/SS/CCCC.h`

database files where SS is the state numeric code and CCCC is the county numeric code.
On the distribution there are maps of the San Francisco area and Manhattan. Look in `/lib/tiger` for the names.

`/lib/tiger/codes1`

is the translation between symbolic names and numeric codes.

BUGS

Tiger takes 16-32 megabytes of memory. It is only installed in `/mips/bin` and should be run only on a large machine.
Because of the large memory usage, it is unwise to display a large number of counties at the same time.
This program is just a toy.

NAME

map, key, plot, photo, movie, report, query, wextract, iupdate − weather maps, reports, photos, and utilities

SYNOPSIS

`weather/photo` [`vis` | `ir`]

`weather/movie` [`vis` | `ir`] [*nobs*]

`weather/map` [`us` | `us24` | `nyc` | `colo` | `dc` | `eur` | `jap` | `radar`]

`weather/key`

`weather/query`

`weather/plot` [*wextract-options*] [*map-location*] [`-r dn`] [`radar`]

`weather/report` [*wextract-options*] [`quake summary`]

`/bin/aux/wextract` [*wextract-options*]

`/bin/aux/iupdate` [`-i` *yyyymm*]

DESCRIPTION

Weather/map displays the latest available national weather map, including precipitation, isobars, fronts, and wind information. The map is updated three times a day. This map is supplied as a free sample from Unidata, and may be terminated for weeks at a time. *Weather/key* displays a reference chart for the symbols in *weather/map*.

Weather/photo fetches and displays the latest satellite photographs. Visible and infrared photographs are selected with `vis` and `ir` respectively. `Ir` is the default. In infrared photos, hotter objects are blacker. The maps and photos are obtained from `vmd.cso.uiuc.edu` which updates them hourly (except visible maps during the hours of darkness) across the entire US. The National Weather Service data is made available courtesy of the National Science Foundation-funded UNIDATA Project and the University of Michigan.

Weather/movie displays the last *nobs* (default 72) weather photos in a loop at five frames per second. Infrared photos are default.

Weather/query connects to the Weather Underground server at the University of Michigan. This is an interactive menu system providing access to a variety of weather information, including forecasts.

Weather/plot plots a weather map. ‘Radar’ displays the precipitation intensities reported in selected time period. The echoes range from 1 (light) to 6 (severe). Snow produces weaker reflections than rain. The `-r` option selects the radar display symbols: `d` for various size dots, `n` for numbers. The default is various shaded circles. For terminals with 1-bit deep bitmaps, the `d` option is default. Radar information is not available outside the US. The time period for the reported information is controlled by *wextract-options* (See below.) The default is the previous 60 minutes.

Map-location selects the area to be displayed:

`e ne se gulf us`	Parts of the US. Default is `e`.
`-L` *lat1 lat2 long1 long2*	Specific latitude and longitude pairs.

Weather/report extracts various reports. Available reports are:

`summary`	A chatty summary of the national weather. Default shows all weather summaries for the last 24 hours.
`quake`	Earthquake reports. Default displays all earthquake reports for the last 72 hours.

Iupdate scans `/lib/weather/raw` and updates `/lib/weather/raw.idx`. This file provides the year and month information (which the `raw` data lacks) and greatly speeds the processing of the huge `raw` file. If the raw file is restarted, the `-i`*yyyymm* option recreates the index file. *Yyyymm* is the year and month of the first record in the raw file.

Wextract prints selected parts of the `raw` file based on time and record type. *Wextract-options* are:

`-a` *age* Maximum age of records before selected time. Default is 1 day.

`-d` *enddate*
Date and time of the last record accepted. Default is the present. Dates may be relative to the current time, or an absolute time. Relative dates begin with a minus and have the form [− *days*.]*hours*[:*minutes* Absolute times have the form [[[*year*/]*month*/]*day*.]*hour*[:*minute*][*TZ*] where *year* is a four digit number and *TZ* is `Z` for GMT or a three-letter time zone identifier.

`-t` *record-type,...*
Types of records selected. Each record in the raw weather file has a three digit type. A record type may be a string of these numbers, or a record name. The records in the FAA-supplied `raw` file are not documented. A listing of the record names appears in the source listing for *wextract*. Some record types are:

`sd` Encoded precipitation radar data.

`wa` AIRMETS for pilots.

`ws` Convective SIGMETS for pilots.

`unknown`
All records not in one of the named types, plus malformed records.

`all`
All records.

FILES

`/lib/weather/raw`	The raw weather data straight off the wire since early June, 1990.
`/lib/weather/raw.idx`	An hourly index into `/lib/weather/raw`.

SEE ALSO

map(7), *plot*(1).

BUGS

Network vagaries and problems on the source machines may make *weather/photo*, *weather/query*, and *weather/map* unavailable or outdated.

The satellite photographs the east coast at an oblique angle.

Iupdate and *wextract* can be slow when the data is WORM-resident, which is likely. Absolute times don't handle time zones correctly. Much of the raw file is a mystery.

The rain echo intensities on *weather/map* do not monotonically darken with increasing precipitation on monochrome displays.

NAME

intro − introduction to system administration

DESCRIPTION

This manual section describes commands for system administration as well as various utility programs necessary for the system but not routinely invoked by a user.

NAME

adduser, changeuser, printnetkey, renameuser, removeuser, enable, disable, expire, status, convkeys, wrkey − maintain authentication databases

SYNOPSIS

`auth/adduser` [`-hnp`] *user*

`auth/changeuser` [`-hnp`] *user*

`auth/printnetkey` *user*

`auth/renameuser` [`-np`] *user newname*

`auth/removeuser` [`-np`] *user*

`auth/enable` [`-np`] *user*

`auth/disable` [`-np`] *user*

`auth/expire` [`-np`] *user date*

`auth/status` *user*

`auth/convkeys` [`-d`] [`-k` *key*] *keyfile*

`auth/wrkey` [`-k` *key*]

DESCRIPTION

These administrative commands run only on the authentication server. *Adduser*, *changeuser*, *renameuser*, *removeuser*, *enable*, *disable*, *expire*, and *status* manipulate an authentication database file system served by *keyfs*(4) and used by file servers. There are two authentication databases, one holding information about Plan 9 accounts and one holding SecureNet keys. A *user* need not be installed in both databases but must be installed in the Plan 9 database to connect to a Plan 9 service.

Adduser installs *user* in an authentication database. *User* must not already exist in the database. It does not install a user on a Plan 9 file server.

Option `-p` installs *user* in the Plan 9 database. *Adduser* asks twice for a password for the new *user.* If the responses do not match or the password is too easy to guess the *user* is not installed.

Option `-n` installs *user* in the SecureNet database and prints out a key for the SecureNet box. The key is chosen by *adduser*.

If neither option `-p` or option `-n` is given, *adduser* installs the *user* in the Plan 9 database.

Option `-h` makes *user* a host able to receive authenticated incoming network calls. All Plan 9 CPU servers must be installed as *users* with host permission in the Plan 9 authentication database. This option is significant only in the Plan 9 database.

Changeuser modifies information for *users* already installed. Its syntax is the same as *adduser's.*

Printnetkey prints *user's* SecureNet key without changing it.

Renameuser changes *user's* name to *newname* in both of the authentication databases. If *newname* is already known in either database, *renameuser* reports an error and makes no change. The options are the same as for *adduser,* except that if neither option `-p` nor option `-n` is given, the user is renamed in both databases.

Removeuser deletes *user* from both of the authentication databases. The options are the same as for *renameuser.*

Enable and *disable* change the status of *user's* accounts. The options are the same as for *renameuser.*

Expire changes the expiration date for *user* to *date*, which is either the string `never` or a date in the form *yyyymmdd*, where *yyyy* is the year, *mm* is the month, and *dd* is the day the account should expire.

Both *enable* and *expire* attempt to change both the Plan 9 and SecureNet databases. The options are the same as for *renameuser.*

Status prints the status and expiration date of *user's* Plan 9 and SecureNet accounts.

Convkeys re-encrypts the key file *keyfile.* Re-encryption is performed in place. Any file or authentication server using the key file must simultaneously have its key modified or it will be unable to decrypt *keyfile.* *Convkeys* uses the key stored in non-volatile RAM to decrypt the file, and encrypts it using the new key. By default, *convkeys* prompts twice for the new password. Option `-k` instead takes *key*, which must be `DESKEYLEN` bytes long. Note that a key is not a password. Option `-d` uses `/dev/crypt` for decrypting the key file. The format of *keyfile* is described in *keyfs*(4).

Wrkey sets the key used by the authentication server to decrypt key files. By default, it prompts twice for the password. Option `-k` is as in *convkeys.* Once the key is set, *keyfs* should be restarted so it serves the correct keys.

FILES

The non-volatile RAM on the server, which stores the key used
to decrypt key files.

SEE ALSO

keyfs(4), *securenet*(8)

BUGS

After changing authentication information, it is necessary to issue the `auth` command on file servers that are doing their own authentication. See *fs*(8).

NAME

boot − connect to the root file server

SYNOPSIS

`/boot` [`-afkmp`] [`-u`*username*] [*method* `!` *fs-addr*]

DESCRIPTION

Boot is the first program run after a kernel has been loaded. It connects to the file server that will serve the root, performs any authentication needed to connect to that server, and *exec*(2)'s the *init*(8) program.

Once loaded, the kernel initializes its data structures and devices. It sets the two environment variables `/env/cputype` and `/env/terminal` to describe the processor. It then binds a place-holder file server, *root*(3), onto `/` and crafts an initial process whose sole function is to *exec*(2) `/boot`, a binary which is compiled into *root*(3).

The command line passed is dependent on the information passed from boot ROM to kernel. On the MIPS Magnum and SGI Power Series the command line passed to *boot* is the same as that given to the ROM monitor.

On AT&T Gnots the command line is

> `/68020/9gnot` *method* `!` *server*

On the Nextstation and the Safari, no information is passed from the boot ROM or program. Their command lines are

```
/68020/9nextstation -p
```

and

```
/386/9safari -p
```

Boot must determine the file server to use and a method with which to connect to it. It must also set a user name to be used as the owner of devices and all console processes and an encryption key to be used when challenged. If the `-m` or `-p` option is given (or the *method* on the command line is invalid) *boot* will prompt for these.

Method and address are prompted for first. The prompt lists all valid methods, the default in brackets.

```
root is from (il, tcp, hs, local)[il]:
```

A newline picks the default. Other possible responses are *method* or *method* `!` *address.*

The other interactions depend on whether the system is a terminal or a CPU server.

Terminal

The terminal must have a *username* to set. If none is specified with the `-u` option, *boot* will prompt for one on the console:

```
user:
```

The user will also be prompted for a password to be used as an encryption key on each *attach*(5):

```
password:
```

With most *methods boot* can now connect to the file server. However, with the serial line *methods* `9600` and `19200`, the actual mechanics of setting up the complete connection are too varied to put into the boot program. Instead *boot* lets the user set up the connection. It prints a prompt on the console and then simulates a dumb terminal between the user and the serial line:

```
Connect to file system now, type ctrl-d when done.
(Use the view or down arrow key to send a break)
```

The user can now type at a modem or a Datakit `destination please:` interface to set up the connection to a TSM8 card. At Murray Hill, a user would type `nj/astro/plan85` at this point. When the user types a control-D, *boot* stops simulating a terminal and starts the file system protocol over the serial line.

Once connected, *boot* `mount`'s the root file system before `/` and makes the connection available as `#s/boot` for subsequent processes to `mount` (see *bind*(2)). *Boot* completes by *exec*(2)'ing `/$objtype/init -t`. If the `-a` or `-m` options are given they are also passed as options to *init*.

CPU Servers

The user owning devices and console processes on CPU servers is always *bootes*. It is immutable. (The name is compiled into the system as the value of the variable `eve`; local sites may choose a different name.) If a `-k` option is given *boot* will prompt for an encryption key to be stored in the CPU server's non-volatile ram.

```
key:
```

This key is used to verify to callers of the CPU server that it is indeed the server being called.

Once connected, *boot* `mount`'s the root file system before `/` and makes the connection available as `#s/boot` for subsequent processes to `mount` (see *bind*(2)). *Boot* completes by *exec*(2)'ing `/$objtype/init -c.` If the `-a` or `-m` options are given they are also passed as options to *init*.

Booting Methods

The methods available to any system depend on what was compiled into the kernel. The complete list of booting methods are listed below.

`cyc`	connect via a point-to-point fiber link using Cyclone boards. If specified, the address must be the number of the Cyclone board to be used, default 0.
`il`	connect via Ethernet using the IL protocol.
`tcp`	connect via Ethernet using the TCP protocol. This method is used only if the initial file server is on a Unix system.
`hs`	connect via Datakit using the high speed Datakit card.
`incon`	connect via Datakit using the Incon interface.
`9600`	connect via Datakit using the serial interface at 9600 baud.
`19200`	connect via Datakit using the serial interface at 19200 baud.
`local`	connect to the local file system.

For the DARPA Internet methods, `il` and `tcp`, the address must be a numeric IP address. If no address is specified a file server address will be found from another system on the network using the BOOTP protocol and the Plan 9 vendor specific fields. For the Datakit methods, `hs`, `9600`, `19200`, and `incon`, the address must be specified and must be a relative path name to the file server. If no address is specified, the address `Nfs` is used.

FILES

`#s/boot`

SEE ALSO

root(3), *bootp*(8), *init*(8)

NAME

booting − bootstrapping procedures

SYNOPSIS

none

DESCRIPTION

This manual collects the incantations required to bootstrap Plan 9 machines. Some of the information here is specific to the installation at Bell Labs; some is generic.

If a CPU server is up, BOOTP and TFTP will run from there; if not, the necessary files and services must be available on a separate machine such as a Unix system to use these protocols for bootstrapping.

Be sure to read *boot*(8) to understand what happens after the kernel is loaded.

Terminals

First, here are instructions to bootstrap the various Plan 9 terminals. To bootstrap a terminal or a CPU server, a file server must be running. On all the terminals, typing two control-T's followed by a lower-case `r` reboots the machine; other methods of rebooting are mentioned for some machines.

Gnot

The boot ROM prints

```
server[default==incon!nj/astro/Nfs!/68020/9gnot]
```

Typing just a newline bootstraps the default system. The components of the `server` string are defaulted from the right, for example, to bootstrap `/sys/src/9/gnot/9gnot` type just that file name; to bootstrap from a different file server, say kremvax, type

```
kremvax!/68020/9gnot
```

The bootstrap devices available are `incon`, `9600`, `19200` and `scsi`; with `scsi` the server name (here `nj/astro/Nfs`) becomes a unit number, usually `0`, and the file name is a boot partition to use. For example,

```
scsi!0!boot
```

means boot from the SCSI disk `0` the kernel in disk partition `/dev/hd0boot`.

If running with a local cache file system, one normally bootstraps using the SCSI disk. However, if the local kernel has been destroyed or is hopelessly out of date, bootstrap using the serial line. To do this, use the boot line

```
9600!nj/astro/Nfs!/68020/9gnotdisk
```

to bootstrap from the serial line at 9600 baud or

```
19200!nj/astro/Nfs!/68020/9gnotdisk
```

for a 19200 baud connection.

To shut the system down, just turn off the power.

Nextstation

When powered on and left alone, a Nextstation will download `/68020/9nextstation` using the BOOTP and TFTP protocols. (Actually, first it loads `/lib/tftpd/boot` and uses that to download the operating system.) It then prompts for the user name and password and asks for the Ethernet protocol to use; request the default.

While the system is downloading, it displays an Ethernet symbol; at this time, holding the left `Command` key down and typing the ~ key aborts the download and transfers control to a ROM-resident monitor. The monitor will use the Ethernet to boot an alternate kernel given the command, e.g.,

```
ben /sys/src/9/next/9nextstation
```

or

```
ben kgbvax:/sys/src/9/next/9nextstation
```

to force the download to come from system `kgbvax`.

If running with a local cache file system, bootstrap from the disk. While the system is downloading, it displays a symbol of a spinning disk. The processor first loads a program, Disclabel, from the kernel partition `/dev/hd1label` and then the real kernel from `/dev/hd1boot`.

See Next's documentation for other details, in particular how to initialize a new machine to boot from Ethernet instead of disk.

To turn the power off, hold down the left `Command` and `Alternate` keys and press the power key. To reboot, hold down the left `Command` and `Alternate` and press the `*` key in the upper left corner of the keypad.

Sun Sparcstation

Type a `b` to the power-on monitor and the kernel will be downloaded. The kernel resides in `/lib/tftpd/`*xxxxxxxx*`.SUN4C` where *xxxxxxxx* is the upper-case hexadecimal IP number of the machine. There is no way to specify an alternate file to download. Once running, the operating system asks the same questions as on the Nextstation.

MIPS Magnum

The Magnum ROM monitor can boot from the ethernet or from a local disk. To boot from the ethernet, type

```
bootp()/mips/9magnum
```

or use the ROM command `setenv` to set the variable `bootfile` to that same string and type `boot`. To load a different file, tell `bootp` which file to load, and to force the download to come from a particular system, `bootp()system:file`. Any arguments after `bootp()file` are passed to `/boot`.

To boot from disk, type

```
dksd()b
```

to load the Plan 9 bootstrap program or use the ROM command `setenv` to set the variable `bootfile` to that same string and type `boot`. The bootstrap program will then prompt for the partition to boot from. If nothing is typed in 15 seconds, a kernel will be booted from the hard disk partition `/dev/hd0boot`. Any arguments after `dksd()b` are passed to `/boot`.

Once running, the operating system asks the same questions as on the Nextstation.

To reboot the machine, cycle the power or hit the reset button on the back.

AT&T Safari and other PCs

The Safari always boots DOS when you turn it on. Once DOS is booted, it will prompt with a `C>`. To that type `b` to boot Plan 9. If there is no diskette in the floppy drive, the kernel will automatically be loaded from the hard disk partition `/dev/hd0boot`. To boot from diskette, insert one (with a DOS file system) before typing `b`. The boot program will prompt for the kernel to boot with

```
server[hd!0!boot]:
```

A return causes a boot from hard disk partition, `/dev/hd0boot`. Typing

```
fd!0!9
```

will bootstrap a kernel from the file named `9` on the diskette.

Once the kernel is booted, it behaves like all others. See *boot*(8) for details.

CPU Servers

The Plan 9 CPU servers are multi-user, so they do not request a user name when booting. On the CPU servers, typing a control-P on the console reboots the machine.

SGI Power Series

To the power-on menu, type a `5` to get the `>>` prompt. Then boot Plan 9 using the Plan 9 bootstrap program, `b`, which resides on the disk volume header (dvh) of the SCSI boot disk. The bootstrap program takes two arguments, the method with which to attach to a file server (as in *boot*(8)) and a kernel file to boot. The default method is *cyc* and the default kernel file is `/mips/9power.` For example, to boot the standard kernel over the Cyclone, just type

```
b
```

To boot a test kernel via the Ethernet using the IL protocol type

```
b il /sys/src/9/power/9power
```

Any arguments given to the bootstrap program will also be passed on to `/boot` in the loaded kernel.

The bootstrap program reads a configuration description from file `/mips/conf/x.x.x.x` where `x.x.x.x` is the decimal value of the each byte of the IP address separated by dots. After loading the kernel it passes to it the configuration information.

Before using Plan 9 for the first time you will have to bring up Unix to copy the bootstrap program to the dvh. Copy `/mips/9powerboot` onto the Unix file system. Then use `dvhtool` to copy it to the dvh file `b.`

Sun Sparcstation

Proceed as for the Sparcstation running as a terminal but load `/sparc/9sscpu`.

Mips Magnum

Booting from the ethernet proceed as for the Magnum running as a terminal but load `/mips/9magnumcpu`. Booting from the disk proceed as for the Magnum running as a terminal.

With a disk a two stage boot is also possible. This allows us to boot from networks and or protocols unknown to the ROM monitor. Instead of copying a kernel into the boot partition, we copy a bootstrap program that will load the real kernel across a network. The arguments following `dksd()b` are passed both to the bootstrap program and to the downloaded kernel. The first specifies the network and machine to boot from and the second specifies the file to boot. For example, to boot via the Datakit high speed board from the file server `nj/astro/x` type

```
dksd()b hs!nj/astro/bootes /mips/9magnumcpu
```

To boot via the ethernet using the IL protocol type

```
dksd()b il /mips/9magnumcpu
```

Hobbit

The ROM uses some variables in non-volatile RAM for booting. The variables can be examined by typing an `e` in response to the ROM prompt `'>>>'` and initialized to useful defaults by `'e -i'`. In order to boot the default kernel (ROM variable `bootfile`)

```
b -- incon!nj/astro/Nfs
```

and to boot an alternate kernel

```
e bootfile /sys/src/9/hobbit/9cpu
b -- incon!nj/astro/Nfs
```

or

```
b /sys/src/9hobbit/9cpu -- incon!nj/astro/Nfs
```

The second form will also initialize the `bootfile` variable.

Currently, Hobbit boards boot only using `incon`.

File servers

The CPU servers and terminals run essentially the same program, but the Plan 9 file servers run a distinct system. The file servers accept only the commands described in *fs*(8) on their consoles.

SGI Power Series

To the power-on menu, type a `5` to get the >> prompt. Then boot the system like a Magnum but load

`9powerfs`. These files are in `/sys/src/fs/power` on `bootes` or in `/usr/local/boot` on `tempel`. The system will come up automatically. On `bootes`, several minutes will be spent initializing the WORM jukebox; the machine will chat happily while this is going on.

Mips 6280

In response to the PROM >> prompt, type

```
bootp(,egl)tempel:96280fs
```

Sparc Sparcstation

Proceed as for the Sparcstation running as a terminal, but load `/sparc/9ssfs`.

Mips Magnum

Proceed as for the Magnum running as a terminal, but load `/mips/9magnumfs`.

SEE ALSO

boot(8), *fs*(8), *init*(8)

BUGS

The file server should be able to boot from its own disk.

NAME

bootp, rarpd, tftpd − Internet booting

SYNOPSIS

`ip/bootp` [`-d`]

`ip/rarpd` [`-d`] [`-e` *etherdev*]

`ip/tftpd` [`-d`] [`-h` *homedir*]

DESCRIPTION

These programs support booting over the Internet. They should all be run on the same server to allow other systems to be booted. *Rarpd* and *tftpd* are used to boot Suns. *Bootp* and *tftpd* are used to boot everything else.

Bootp passes to Plan 9 systems their IP address, IP mask, default boot file, default file server, default authentication server, and default gateway. These come from the network database file attributes *ip*, *ipmask*, *bootf*, *fs*, *auth*, and *ipgw* attributes respectively (see *ndb*(6) and *ndb*(8)). The attributes come from the entry for the system, its subnet, and its network with the system entry having precedence, subnet next, and network last. The `-d` option causes debugging to be printed to standard out.

Rarpd performs the Internet reverse address resolution protocol, translating Ethernet addresses into Internet addresses. The options are:

`d`	print debugging to standard output
`e`	use the Ethernet mounted at /net/*etherdev*

Tftpd transfers files to systems that are booting. It runs as user *none* and can only access files with world read permission. The options are:

`d`	print debugging to standard output
`h`	change directory to *homedir*. The default is `/lib/tftpd.` All requests for files with non-rooted file names are served starting at this directory with the exception of files of the form `xxxxxxxx.SUN4C.` These are Sparc kernel boot files where `xxxxxxxx` is the hex IP address of the machine requesting the kernel. *Tftpd* looks up the file in the network database using and responds with the bootfile specified for that particular machine. If no booftile is specified, the transfer fails. *Tftpd* supports only octet mode.

SEE ALSO

ndb(6)

NAME

btrace − trace bitblt protocol

SYNOPSIS

`btrace` [`-d` [`d`]] [`-o` *ofile*] [`-b` *bfile*]

DESCRIPTION

Btrace eavesdrops on messages to and from `/dev/bitblt`, interprets them as messages in the bit device protocol (see *bit*(3)), and prints a readable version of the messages on a trace file, `btrace.out` by default. After *btrace* is started, it runs in the background, tracing all subsequent graphics programs run in that window. Options for *btrace* are:

`-o` *ofile* Print trace output in *ofile* instead of `btrace.out`.

`-b` *bfile* Dump each bitmap read or written to *bfile*, using the format of *bitmap*(6). Each succeeding bitmap overwrites the previous contents of *file*. *Tweak*(1) can be used to examine the file.

`-d` Increase the level of trace detail. The maximum level is `-dd`.

FILES

`/dev/bitblt`

SEE ALSO

bit(3), *bitmap*(6), *graphics*(2), *tweak*(1)

NAME

cpurc, termrc − boot script

SYNOPSIS

`cpurc`

`termrc`

DESCRIPTION

After the kernel boots, it execs `/boot` (see *root*(3)), which in turn execs `/$cputype/init`. *Init*(8) sets the `$service` environment variable to `cpu` or `terminal`, and then invokes the appropriate `rc` script to bring the system up.

Based on the values of `$sysname` and `$terminal` these scripts start appropriate network processes and administrative daemons and enable swapping. `Cpurc` sets `/env/boottime` to the time *cpurc* was executed and `/env/NPROC` to a value suitable for parallel compilation in *mk*(1).

SEE ALSO

srv(4), *namespace*(6), *dkconfig*(8), *init*(8), *listen*(8)

NAME

cron − clock daemon

SYNOPSIS

`auth/cron` [`-c`]

DESCRIPTION

Cron executes commands at specified dates and times according to instructions in the files `/cron/`*user*`/cron.` It runs only on an authentication server. Option `-c` causes *cron* to create `/cron/`*user* and `/cron/`*user*`/cron` for the current user; it can be run from any Plan 9 machine.

Blank lines and lines beginning with `#` in these files are ignored. Entries are lines with fields

minute hour day month weekday host command

Command is a string, which may contain spaces, that is passed to an *rc*(1) running on *host* for execution. The first five fields are integer patterns for

minute	0−59
hour	0−23
day of month	1−31
month of year	1−12
day of week	0−6; 0=Sunday

The syntax for these patterns is

```
time  : '*'
      | range
range : number
      | number '-' number
      | range ',' range
```

Each number must be in the appropriate range. Hyphens specify inclusive ranges of valid times; commas specify lists of valid time ranges.

Cron is not a reliable service. It skips commands if it cannot reach *host* within two minutes, or if the *cron* daemon is not running at the appropriate time.

EXAMPLES

Here is the job that mails system news.

```
% cat /cron/upas/cron
# send system news
15 8-17, 21 ***   helix /mail/lib/mailnews
%
```

SEE ALSO

con(1), *rc*(1)

NAME

dkconfig, dkstat − configure Datakit interface

SYNOPSIS

`dkconfig` [`-d` *dev*] [`-n` *netname*] [`-c` *csc nlines*] [`-b` *baud*] [`-w` *window*] [`-ai`]

DESCRIPTION

Dkconfig configures the device *dev* (default `#h`) as a Datakit link and gives it kernel id *netname* (default `dk`). Any subsequent reference to the device `#k`*name* and its subdirectories refers to conversations multiplexed on this link.

As a convenience, *dkconfig* performs a

```
bind("#kname", "/net", MBEFORE)
```

to make the dk device available to *dial*(2).

Option `-c` allows the common signalling channel, *csc,* and the number of Datakit lines, *nlines*, to be specified.

Option `-w` sets the window size to *window,* a decimal number of bytes. This is most important on the Safari's Incon interface which overflows if the window size is greater than 256.

Option `-i` causes an incon device (default `#i`) to be configured as the Datakit connection.

Option `-a` causes the a serial line (default `/dev/eia0`) to be configured as the Datakit connection. The `async` line protocol is pushed onto the serial line's stream to provide a multiplexed connection.

FILES

`#h`	default device
`#i`	incon device
`#k*/dk/*`	Datakit devices
`/net/dk`	by convention, Datakit device bind point

SEE ALSO

listen(8), *datakit*(3), *dk*(3), *dial*(2) *netstat(1)*

NAME

fs, exsort − file server maintenance

SYNOPSIS

`help` [*command ...*]
`arp` *subcommand*
`auth` [`on`] [*system* | *file*]
`cfs` *filesystem*
`check` [*options*]
`clri` [*file*...]
`cpu` [*proc*]
`create` *path uid gid perm* [`lad`]
`cwcmd` *subcommand*
`cycl` *subcommand*
`date` [[`+−`] *seconds*]
`dump`
`flag` *flag* [*channel*]
`halt`
`netdb` [*file*]
`newuser` *name* [*options*]
`passwd`
`profile` [`01`]
`remove` [*files*...]
`search` [*blockno* [*nblock* [`bw`]]]
`stat[acejklw]`
`statp` [*proc*]
`stats` [[`−`] *flags*...]
`sync`
`time` *command*
`trace` [*number*]
`users` [*file*]
`version`
`who` [*user*...]
`wormcp` [*funit tunit* [*nblock*]]

`disk/exsort` [`-w`] [*file*]

DESCRIPTION

Except for *exsort,* these commands are available only on the console of an *fs*(4) file server.

Help prints a ‘usage string’ for the named *commands,* by default all commands. Also, many commands print menus of their options if given incorrect or incomplete parameters.

The console requires the machine’s password to be supplied before accepting commands. Typing a control-D will cause the server to request the password again.

Arp has two *subcommands:* `print` prints the contents of the ARP cache and `flush` flushes it.

Auth starts authentication. It reads authentication keys from *file* (default `/adm/keys`). If *file* is of the form `il!`*IP.address* it is taken to be a *system* from which to authenticate using IL, rather than reading its own keys file. The address must be a numeric IP address and only IL is supported. If the optional string `on` is provided, *auth* records in non-volatile RAM the *file* or *system* from which to authenticate. Once on, authentication can never be turned off, even by rebooting.

Cfs changes the current file system, that is, the file tree to which commands (`check`, `clri`, `create`, `netdb`, `newuser`, `profile`, `remove`, and `users`) apply. The initial *filesystem* is `main`.

Check verifies the consistency of the current file system. With no options it checks and reports the status. It suspends service while running. Options are:

`rdall`	Read every block in the file system (can take a *long* time).
`tag`	Fix bad *tags*; each block has a tag that acts as a backwards pointer for consistency checking.
`pfile`	Print every file name (can take a *long* time).
`free`	Rebuild the list of free blocks.
`setqid`	Resequence the qids in the file system, starting at one; all outstanding fids become invalid.
`bad`	For each block with a bad tag, create a new block, copy the data from the bad block, and write the correct tag in the new block.
`touch`	Cause every directory and indirect block not on the current WORM disk to be advanced to the current WORM on the next dump.

Clri clears the internal directory entry and abandons storage associated with *files.* It ignores the usual rules for sanity, such as checking against removing a non-empty directory. A subsequent `check free` will place the abandoned storage in the free list.

Cpu prints the CPU utilization and state of the processes in the file server.

Create creates a file on the current file system. *Uid* and *gid* are names or numbers from `/adm/users`. *Perm* is the low 9 bits of the permission mode of the file, in octal. An optional final `l`, `a`, or `d` creates a locked file, append-only file, or directory.

Cwcmd controls the cached WORM file systems.

`mvstate` *state1 state2*

States are `none`, `dump`, `dump1`, `read`, and `write`. A `mvstate dump1 write` will cause I/O errors in the last dump to be retried in the next dump. A `mvstate read none` will flush the cache associated with the WORM. A `mvstate dump write` aborts the background process dumping to WORM; as a consequence it leaves holes in the dump file system. Other uses are possible but arcane.

`prchain` [*start*] [`back`]

Print the chain of superblocks for the directory containing the roots of the dumped file systems, starting at block number *start* (default 0) going forward (backwards if `back` is supplied).

`savecache`

Copy the block numbers, in native endian longwords, of all blocks in the `read` state to the file `/adm/cache` for use by `disk/exsort`.

`loadcache` [*dskno*]

Read `/adm/cache` and for every block there on WORM disk *dskno*, read the block from WORM to the cache. If *dskno* is not supplied, all blocks in `/adm/cache` are read.

`wormcmp [`*dskno*`]`

Read WORM disk *dskno* and compare it to the contents of the cache, block by block. *Dskno* is zero by default.

`startdump [01]`

Suspend (`0`) or restart (`1`) the background dump process.

Cycl controls the Cyclone fiber link to the main CPU server. The subcommands are

`reboot`	Reinitialize the Cyclone board and connection.
`verbose`	Put the Cyclone driver in verbose debugging mode.
`ping`	Bounce a packet off the remote Cyclone board; used internally to resynchronize after an error on the fiber.

Date prints the current date. It may be adjusted using `+-`*seconds.* With no sign, it sets the date to the absolute number of seconds since 00:00 Jan 1, 1970 GMT; with a sign it trims the current time.

Dump starts a dump to WORM immediately for all file systems that have a WORM associated. File service is suspended while the cache is scanned; service resumes when the copy to WORM starts.

Flag toggles flags, initially all off:

`arp`	Report ARP activity.
`attach`	Report as connections are made to the file server.
`chat`	(Very noisy.) Print all 9P messages to and from the server.
`dkit`	Report datakit activity.

If given a second numeric *channel,* as reported by *who,* the flag is altered only on that connection.

Halt does a `sync` and halts the machine, returning to the boot ROM.

Netdb reads `/lib/ndb/local` to establish network information.

Newuser requires a *name* argument. With no options it adds user *name,* with group leader *name,* to `/adm/users` and makes the directory `/usr/`*name* owned by user and group *name.* The options are

`?`	Print the entry for *name.*
`:`	Add a group: add the name to `/adm/users` but don't create the directory. By convention, groups are numbered starting from 10000, users from 0.
newname	Rename existing user *name* to *newname.*
=*leader*	Change the leader of *name* to *leader.* If *leader* is missing, remove the existing leader.
+*member*	Add *member* to the member list of *name.*
−*member*	Remove existing *member* from the member list of *name.*

After a successful *newuser* command the file server overwrites `/adm/users` to reflect the internal state of the user table.

Passwd sets the machine's password and writes it in non-volatile RAM.

Profile `0` clears the profiling buffer and enables profiling; *profile* `1` stops profiling and writes the data to `/adm/kprofdata` for use by `kprof` (see *prof*(1)). If a number is not specified, the profiling state toggles.

Remove removes *files.*

Search looks on the WORM for written (`w`; default) or blank (`b`) blocks starting at block *blockno* (default 0) through *nblock* (default 100) following blocks. Block numbers are as reported by *statw.*

The *stat* commands are connected with a service or device identified by the last character of the name: `c`, Cyclone fiber link; `e`, Eagle Ethernet controller; `j`, Jaguar SCSI/VME disk controller; `k`, Datakit; `l`, LANCE Ethernet controller; `w`, cached WORM. The *Statp* command prints statistics about processes; an optional argument identifies the process to be displayed; *stata* prints overall statistics about the file system. The *stats* command takes an optional argument identifying the characters of *stat* commands to run. The option is remembered and becomes the default for subsequent *stats* commands if it begins with a minus sign.

Sync writes dirty blocks in memory to the magnetic disk cache.

Time reports the time required to execute the *command.*

Trace with no options prints the set of queue-locks held by each process in the file server. If things are quiescent, there should be no output. With an argument *number* it prints a stack traceback of that process.

Users uses the contents of *file* (default `/adm/users`) to initializes the file server's internal representation of the users structure. Incorrectly formatted entries in *file* will be ignored. If file is explicitly `default`, the system builds a minimal functional users table internally; this can help recover from disasters. If the *file* cannot be read, you *must* run

```
users default
```

for the system to function. The `default` table looks like this:

```
-1:adm:adm:
0:none:adm:
1:rob:rob:
10000:sys::
10001:map:map:
10002:doc::
10003:upas:upas:
10004:cda::
10005:bootes:bootes:
```

Version reports when the file server was last compiled and last rebooted.

Who reports, one per line, the names of users connected to the file server and the status of their connections. The first number printed on each line is the channel number of the connection. If *users* are given the output selects connections owned by those users.

Wormcp copies from WORM disk *funit* to WORM disk *tunit nblock* native blocks (default the whole disk). If *tunit* is written, *wormcp* guarantees the written data is equal to the data on *funit* and stops if not. *Wormcp* does a binary search to find the lowest unwritten block on *tunit* at which to start the copy. With no arguments, *wormcp* stops a running copy.

When the file server boots, it prints the message

```
for config mode hit a key within 5 seconds
```

If a character are typed within 5 seconds of the message appearing, the server will enter config mode. See *fsconfig*(8) for the commands available in config mode. The system also enters config mode if, at boot time, the non-volatile RAM does not appear to contain a valid configuration.

Exsort is a regular command to be run on a CPU server, not on the file server console. It reads the named *file* (default `/adm/cache`) and sorts the cache disk block numbers contained therein. It assumes the numbers are 4-byte integers and guesses the endianness by looking at the data. It then prints statistics about the cache. With option `-w` it writes the sorted data back to *file.*

SEE ALSO

fs(4)

NAME

fsconfig − configuring a file server

SYNOPSIS

`service` *name*

`config` *device*

`filsys` *name device*

`ream` *name*

`recover` *name*

`ip` *ipaddr*

`ipgw` *ipaddr*

`ipmask` *ipaddr*

`ipauth` *ipaddr*

`end`

DESCRIPTION

When a file server's configuration has not been set, or by explicit request early in the server's initialization (see *fs*(8)), the server enters 'config mode'. The commands described here apply only in that mode. They establish configuration constants that are typically valid for the life of the server, and therefore need be run only once. If the non-volatile RAM on the server gets erased, it will be necessary to recreate the configuration.

In these commands, *ipaddr* is an IP address in the form `111.103.94.19` and *name* is a text string without white space. The syntax of a *device* is more complicated:

`w.`*n1*`.`*n2*`.`*n3*
A SCSI disk on target id *n2*, unit *n1*, and partition *n3*. The values *n1* and *n3* (and their associated periods) are optional; they default to zero. Any one of the numbers may be replaced by `<`*m*`−`*n*`>` to represent the values *m* through *n* inclusive. For example, `(w<1-4>)` is the concatenation of SCSI targets 1 through 4.

`r.`*n1*`.`*n2*`.`*n3*
A SCSI WORM disk on unit *n1*, target *n2*, and partition *n3*. The values are as in `w`.

`(`*device...*`)`
A pseudo-device formed from the concatenation of the *devices* in the list. The devices are *not* blank- or comma-separated.

`[`*device...*`]`
A pseudo-device formed from the block-wise interleaving of the *devices* in the list. The size of the result is the number of devices times the size of the smallest device.

`p`*device*`.`*n1*`.`*n2*
A partition starting at *n1*% from the beginning of *device* with a length *n2*% of the size of the device. Parenthesize *device* if it contains periods.

`f`*device* A pseudo-WORM disk: blocks on *device* can be written only once and may not be read unless written.

`c`*device1device2*
A cached WORM. The first *device* is the cache, the second the WORM.

`o` (Letter o) The read-only (dump) file system of the previously defined cached WORM file system.

The `service` command sets the textual name of the server as known in the network databases.

The configuration information is stored in block zero on a device whose device string is written in non-volatile RAM. The `config` command identifies the *device* on which the information is recorded.

The *filsys* command configures a file system on *device* and calls it *name. Name* is used as the specifier in `attach` messages to connect to that file system. (The file system *main* is the one attached to if the specifier is null; see *attach*(5)).

The *ream* command initializes the named file system. It overwrites any previous file system on the same device and creates an empty root directory on the device. If *name* is `main`, the file server, until the next reboot, will accept `wstat` messages (see *stat*(5)) that change the owner and group of files, to enable initializing a fresh file system from a *mkfs*(8) archive.

`recover` *name*
: The named file system must be a cached WORM. *Recover* clears the associated magnetic cache and initializes the file system, effectively resetting its contents to the last dump.

The rest of the commands record IP addresses: the file server's address (*ip*), the local gateway's (*ipgw*), the local authentication server's (*ipauth*), and the local subnet mask (*ipmask*). *Ipauth* should be `0.0.0.0` if the system is doing its own authentication rather than calling an external authentication server.

The various configuration commands only record what to do; they write no data to disk. The command *end* exits config mode and begins running the file server proper. The server will then perform whatever I/O is required to establish the configuration.

EXAMPLE

Initialize a file server `kgbsun` with a single file system interleaved between SCSI targets 3 and 4.

```
service kgbsun
config w3
filsys main [w<3-4>]
ream main
```

Initialize a file server `kremvax` with a single disk on target 0 partitioned as a cached pseudo-WORM file system with the the cache on the third quarter of the drive and the pseudo-WORM on the interleave of the first, second, and fourth quarters.

```
service kremvax
config p(w0)50.1
filsys main cp(w0)50.25f[p(w0)0.25p(w0)25.25p(w0)75.25]
filsys dump o
ream main
```

BUGS

NAME

home, 40meg, 80meg, 100meg, newkernel, personalize, update, Disclabel − administration for local file systems

SYNOPSIS

```
gnot/home
gnot/personalize
gnot/update

magnum/home
magnum/personalize
magnum/update

safari/40meg
safari/80meg
safari/personalize
safari/update

nextstation/100meg
nextstation/personalize
```

```
nextstation/update
disk/newkernel
cp Disclabel /dev/hd1label
```

DESCRIPTION

These programs help maintain a file system on a local disk for a private machine.

Home partitions a disk, copies the appropriate kernel to the disk, and makes a new file system on the disk. *40meg*, *80meg*, and *100meg* configure disks and make file systems for disks of the appropriate size.

Update copies the current kernel to the disk and updates files on the local file system. It only updates those files put there by the *home* program.

Personalize removes the contents of the `/usr` directory on the local disk and copies a minimal set of files for the user who runs the command.

Newkernel updates the kernel in the boot partition. If the running kernel was linked more than 10 minutes before the kernel on the file system was installed, *newkernel* verifies with the user that the kernel should be installed, and copies the kernel.

Disclabel is the bootstrap program copied into the partition `/dev/hd1label` on Nextstations.

FILES

`/lib/proto/portproto`
Mkfs prototype files for *magnum/home*, *magnum/update*, *gnot/home*, and *gnot/update*.

`/lib/proto/386proto`
prototype files for *Mkfs safari/40meg*, *safari/80meg*, and *safari/update*.

SEE ALSO

kfs(4), *mkfs*(8), *prep*(8), *hard*(3)

NAME

init − initialize machine or connection

SYNOPSIS

`/$cputype/init` [`-ctm`] [*command* ...]

DESCRIPTION

Init initializes the machine: it establishes the name space (see *namespace*(4) and *newns* in *auth*(2)), and environment (see *env*(3)) and starts a shell (*rc*(1)) on the console. If a *command* is supplied, that is run instead of the shell. On a CPU server the invoked shell runs *cpurc*(8) before accepting commands on the console; on a terminal, it runs *termrc* and then the user's profile. Options `-t` (terminal) and `-c` (CPU) force the behavior to correspond to the specified service class. Otherwise *init* uses the value of the environment variable `$service` to decide the service class.

Init sets environment variables `$service` (either to the incoming value or according to `-t` or `-c`), `$objtype` (to the value of `$cputype`), `$user` (to the contents of `#c/user`), and `$timezone` (to the contents of `/adm/timezone/local`).

With option `-m` *init* starts only an interactive shell regardless of the *command* or service class.

On a CPU server, *init* requires the machine's password to be supplied before starting *rc* on the console.

Init is invoked by *boot*(8), which sets the arguments as appropriate.

SEE ALSO

rc(1), *auth*(2), *boot*(8)

NAME

ipconfig, arpd − Internet configuration

SYNOPSIS

`ip/ipconfig` [`-ap`] [`-e` *etherdev*] [`-m` *ip-mask*] [`ipaddr`]

`ip/arpd` [`-pd`] [`-e` *etherdev*] [`-b` *bcast-addr*]

DESCRIPTION

Ipconfig configures an Internet connection on an Ethernet. The options are

`a`	do not start *arpd*
`p`	start *arpd* in promiscuous mode (see below)
`e`	use the Ethernet mounted at /net/*etherdev*
`m`	set the network mask to *ip-mask*

If *ipaddr* is specified on the command line, use that instead of one found int the local database or via the Bootp protocol.

Arpd performs the Internet Address Resolution Protocol, translating Internet addresses into Ethernet addresses. It is normally started by *ipconfig*. The options are

`d`	print debugging to standard output
`p`	(promiscuous) answer ARP requests for any recognized machine. The default is to answer just for the machine running *arpd.*
`e`	use the Ethernet mounted at /net/*etherdev*
`b`	use the IP broadcast address *bcast-addr* instead of the correct one.

SEE ALSO

ndb(6)

NAME

kfscmd, ksync − kfs administration

SYNOPSIS

`disk/kfscmd` [`-n` *name*] cmd ...

`disk/ksync`

DESCRIPTION

Kfs is a local user-level file server for a Plan 9 terminal with a disk. *Kfscmd* transmits commands to the *kfs* server (see *kfs*(4)). The `-n` option changes the name of the kfs service to `kfs.`*name* (by default, full name is just `kfs`).

Ksync executes the `sync` command for all active *kfs* servers.

The known commands are described below. Note that some commands are multiple words and should be quoted to appear as a single argument to *rc*(1).

`allow`	Turn permission checking off (to simplify administration).
`disallow`	Turn permission checking on.
`halt`	write all changed blocks and stop the file system.
`help`	print the list of commands.
`rename` *file name*	Change the name of *file* to *name*.
`newuser` *user*	Add *user* to `/adm/users` and make the standard directories needed for booting.
`remove` *file*	Remove *file* and place its blocks on the free list.
`clri` *file*	Remove *file* but do not place the blocks on the free list. This command can be used to remove files that have duplicated blocks. The non-duplicate blocks can be retrieved by checking the file system with option `f` (see below).
`create file owner group mode [adl]`	Create the file. Owner and group are users in `/adm/users` and mode is an octal number. If present, `a` creates an append only file, `d` creates a directory, and `l` creates a file that is exclusive-use.
`sync`	write to disk all of the dirty blocks in the memory cache.
`check [PRdfprtw]`	Check the file system. The options are

`p`	print the names of directories as they are checked.
`P`	print the names of all files as they are checked.
`r`	read all of the data blocks and check the tags.
`f`	rebuild the list of free blocks.
`d`	delete redundant references to a block.
`t`	fix bad tags.
`c`	fix bad tags and clear the contents of the block.
`w`	write all of the blocks that are touched.

SEE ALSO

kfs(4), *mkfs*(8), *prep*(8), *hard*(3)

BUGS

Unreliable and unsafe.

NAME

listen, dkcpu, dkcpunote, dkdiscard, dkecho, dkexportfs, dkexportfs0, dkrexexec, dkwhoami, dksmtp, dkdcon, dklogin, dkfsauth, dkrexauth, dkchal, dkchangekey, dkcheck, dkguard, il7, il9, il565, il17005, il17006, il17007, il17009, il17020, il17021, il17022, il17023, il17024, tcp7, tcp9, tcp21, tcp23, tcp25, tcp513, tcp564, tcp565, tcp17007, tcp6000 − listen for calls on a network device

SYNOPSIS

`aux/listen` [`-q`] [`-d` *srvdir*] [`-t` *trustsrvdir*] [*net* [*name*]]

DESCRIPTION

listen announces itself to a network as *name* (by default the contents of `/env/sysname`) and listens for inbound calls to local services. *Net* is the network device on which to listen, by default `/net/dk`. The services available are executable files in *srvdir* or *trustsrvdir*. If neither *srvdir* nor *trustsrvdir* is given, *listen* looks for executable files in `/bin/service`. Services found in *srvdir* are executed as user `none`; services found in *trustsrvdir* as executed as the user who started *listen.* Option `-q` suppresses affirmative log information.

Service names are made by concatenating the name of the network with the name of the service or port. For example, an inbound call on the TCP network for port 565 executes service `tcp565`, while a call on the Datakit network for service whoami executes service dkwhoami. Services are executed with the name of the service, the network name, and the stream directory of the incoming call as arguments.

The following services are available in `/bin/service`.

`dkcpu il17005`	server for *cpu*(1) command.
`dkcpunote il17006`	`/proc/`*pid*`/notify` forwarding for *cpu*(1).
`dkexportfs il17007 tcp17007`	serve a piece of the name space using the Plan 9 file system protocol, with authentication (typically used by *cpu*(1)).
`dkexportfs0 il17008 tcp564`	like 17007, without authentication (used by Unix systems to see Plan 9 files).
`dkrexexec il17009`	remote execution.
`dkwhoami il565 tcp565`	report the address of the incoming call.
`tcp21`	FTP daemon
`tcp6000`	X-window callback
`dksmtp tcp25`	mail delivery.
`dkdcon`	research Unix terminal connection.
`dklogin`	generic terminal connection.
`tcp23`	telnet terminal connection.
`tcp513`	rlogin terminal connection.
`dkecho il7 tcp7`	echo any bytes received (bit mirror)
`dkdiscard il9 tcp9`	consume any bytes received (bit bucket)

The following services are available in `/bin/service.auth`.

`dkfsauth il17020`	file system authentication.
`dkrexauth il17021`	remote execution authentication.
`dkchangekey il17022`	change a user's password.
`dkchal il17023`	SecureNet CPU authentication.
`dkcheck il17024`	check a user's password.
`dkguard`	check a SecureNet box.

FILES

`/net/dk`	by convention, Datakit device bind point
`/net/il`	by convention, IL device bind point
`/net/tcp`	by convention, TCP device bind point

`/env/sysname`	default announced name

SEE ALSO

dkconfig(8), *auth*(6), *dk*(3), *dial*(2)

NAME

login − set user name

SYNOPSIS

`login` [-u *user*] [cmd]

DESCRIPTION

Login establishes a new name space for a new user and runs a command in that environment. *Login* fist asks for a user name and challenges the user with a string appropriate for use with a Digital Pathways SecureNet encryption box. A fresh name space is constructed for the user and *rc* is started. If *cmd* is given, the arguments `−c` *cmd* are passed to *rc*; otherwise, an interactive *rc* is started.

The option `−u` starts the process as *user* and suppresses the user name prompt.

SEE ALSO

rc(1), *auth*(8)

NAME

lp − PostScript preprocessors

DESCRIPTION

These files reside in `/sys/lib/lp/process` and provide an interface to PostScript conversion programs that can be found in `/sys/lib/postscript/bin/$cputype`. These preprocessors may be selected with the `−p`*process* After each processor description, there is a list of *lp* options to which the processor responds.

generic	is the default preprocessor. It uses *file*(1) to determine the type of input and executes the correct preprocessor for a given (input, printer) pair.
post	passes PostScript through adding option patches for paper tray information. This does not always work with PostScript generated on other systems.
noproc	passes files through untouched.
ppost	converts a text file to PostScript. `[DLcfilmnorxy]`
dpost	converts a troff output file to PostScript. `[DLcimnorxy]`
dvipost	converts a TeX output file to PostScript. `[Lcinor]`
p9bitpost	converts a Plan 9 bitmap (i.e. `/dev/screen`, `/dev/window`, `/dev/windows/*/window`) to PostScript. `[Lm]`
tpost	converts Tektronix 4014 plot codes to PostScript. `[Lcimnorxy]`
hpost	adds a header page to the beginning of a PostScript printer job so that it may be separated from other jobs in the output bin. The header has the image of the jobs owner from the directory of faces. Page reversal is also done in this processor.

SEE ALSO

lp(1)

BUGS

The *file* command is not always smart enough to deal with certain file types. There are PostScript conversion programs in `/sys/lib/postscript/bin/$cputype` that do not have preprocessors to drive them.

NAME

mk9660, pump − create and write ISO-9660 CD-ROM images

SYNOPSIS

`disk/mk9660` [`-c`] [`-e`] [`-a` *absfile*] [`-b` *bibfile*] [`-n` *notfile*] [`-o` *ofile*] *ifile*

`disk/pump` [`-t` *target*] [`-m` *meg*] [`-n` *nproc*] [*ifile*]

DESCRIPTION

Mk9660 reads the file system archive *ifile* as prepared by *mkfs*(8) and produces a file system on *ofile* (`cd-rom` by default) in ISO-9660 format.

The options to *mk9660* are:

`c` Convert all file names in the file system so that they conform to 9660 standards. (Roughly this is eight or fewer single case alphanumerics followed by an optional period and three or fewer single case alphanumerics.) File names that conform are converted from lower case letters in the input filesystem to upper case in the output file system. Files that do not conform are renamed to `F`*number* and directories are renamed `D`*number.* A file named `_CONFORM.MAP` is created in the root of the output file system with old-name new-name pairs of all converted files.

`e` Add a *system-use* field to every directory record that contains the name, uid, gid and mode of the file. With or without this extension, directory records conform to the 9660 standard and should be able to be read on other systems.

`a` Places the named file to the abstract field of the primary volume descriptor. The file must be in the root directory.

`b` Places the named file in the bibliographic field of the primary volume descriptor. The file must be in the root directory.

`n` Places the named file in the copyright field of the primary volume descriptor. The file must be in the root directory.

All dates in the output file system are set to the date the command was executed. The volume identifier field of the primary volume descriptor is set to the last component of *ifile*. The system identifier field of the primary volume descriptor is set to `PLAN 9`, and should be keyed to the interpretation of the system-use fields of the directory records.

Pump reads *ifile* (`cd-rom` by default) and issues the SCSI commands to write a Phillips CDD 521 Compact Disk Recorder. The file is uninterpreted, but it can be a 9660 file system as produced by *mk9660*(8). The CD writer requires a sustained data rate of 305,600 bytes/sec. To get this rate on an Ethernet from a file server, *pump* creates several processes that read ahead into shared buffers. Even so, the file system should have fast disks or multiple disks with interleaved overlapping seeks. The options to *pump* are:

`t` specifies the SCSI target (default 1) for the CD writer.

`m` specifies the total buffer space (default 10) in megabytes.

`n` specifies the number (default 3) of read-ahead processes.

BUGS

The *pump* command does not correctly set up the SCSI bus. Use the *scuzz*(8) command to *open* the desired SCSI target.

SEE ALSO

scsi(3), *mkfs*(8), *scuzz*(8).

NAME

mkfs, mkext, flio − archive or update a file system

SYNOPSIS

`disk/mkfs` [`-aprv`] [`-n` *name*] [`-s` *source*] [`-u` *users*] [`-z` *n*] *proto* ...

`disk/mkext` [`-d` *name*] [`-u`] [`-h`] *file* ...

`disk/flio` [`-io`] [`-b` *bsize*] *diskfile* ...

DESCRIPTION

Mkfs copies files from the file tree *source* (default `/`) to a `kfs` file system (see *kfs*(4)). The kfs service is mounted on `/n/kfs`, and `/adm/users` is copied to `/n/kfs/adm/users`. The *proto* files are read, and any files specified in them that are out of date are copied to `/n/kfs`.

Each line of the *proto* file specifies a file to copy. Indentation is significant, with each level of indentation corresponding to a level in the file tree. Fields within a line are separated by white space. The first field is the last path element in the destination file tree. The second field specifies the permissions. The third field is the owner of the file, and the fourth is the group owning the file. The fifth field is the name of the file from which to copy; this file is read from the current name space, not the source file tree. All fields except the first are optional.

Names beginning with a `$` are expanded as environment variables. If the first file specified in a directory is `*`, all of the files in that directory are copied. If the first file is `+`, all of the files are copied, and all subdirectories are recursively copied.

Mkfs copies only those files that are out of date. Such a file is first copied into a temporary file in the appropriate destination directory and then moved to the destination file. Files in the *kfs* file system that are not specified in the *proto* file are not updated and not removed.

The options to *mkfs* are:

`a` Instead of writing to a `kfs` file system, write an archive file to standard output, suitable for *mkext.* All files in *proto,* not just those out of date, are archived.

`n` *name*
Use kfs.*name* as the name of the kfs service (default `kfs`).

`p` Update the permissions of a file even if it is up to date.

`r` Copy all files.

`s` *source*
Copy from files rooted at the tree *source*.

`u` *users*
Copy file *users* into `/adm/users`.

`v` Print the names of all of the files as they are copied.

`z` *n* Copy files assuming kfs block *n* (default 1024) bytes long. If a block contains only 0 bytes, it is not copied.

Mkext unpacks archive files made by the `-a` option of *mkfs.* The `-d` option specifies a directory to serve as the root of the unpacked file system. The `-u` option, to be used only when initializing a new *fs*(4) file system, sets the owners of the files created to correspond to those in the archive. (This is only permitted at the initial load of the files into a file system.) Each file on the command line is unpacked in one pass through the archive. If the file is a directory, all files and subdirectories of that directory are also unpacked. When a file is unpacked, the entire path is created if it does not exist. If no files are specified, the entire archive is unpacked; in this case, missing intermediate directories are not created. The `-h` option prints headers for the files on standard output instead of unpacking the files.

Flio allows multiple floppy disks to be treated as a single volume. With the `-i` option *flio* reads consecutive floppies from the device file, *diskfile*, and writes the contents to standard output. With the `-o` option *flio* reads from standard input and writes to *diskfile*. The user is prompted whenever a new disk needs to be inserted. The `-b` option specifies a blocking factor to be used on the floppy. *Bsize* is a number of bytes (default 1024) or, with a trailing `k`, a multiple of 1024 bytes.

EXAMPLES

Make an archive to establish a new file system:

```
disk/mkfs -a -u files/adm.users -s dist proto > arch
```

Unpack that archive onto a new file system:

```
srv il!newfs
mount -c /srv/il!newfs /n/newfs
disk/mkext -u -d /n/newfs < arch
```

Unpack an archive from a set of floppy disks:

```
srv il!newfs
mount -c /srv/il!newfs /n/newfs
disk/flio -b 32k -i /dev/fd0disk | disk/mkext -u -d /n/newfs
```

FILES

`/lib/proto/portproto`
generic prototype file.

`/lib/proto/cdaproto`
prototype file for cda programs and libraries.

SEE ALSO

prep(8), *kfscmd*(8), *hard*(3)

NAME

aux/mouse − configure a mouse to a port

SYNOPSIS

`aux/mouse` *port*

DESCRIPTION

`Mouse` queries a mouse on a serial or PS2 port for its type and then configures the port and the mouse to be used to control the cursor.

Port can be either a port number (e.g. `0` or `1`) or the string `ps2`.

SEE ALSO

cons(3)

NAME

query, mkhash, mkdb, cs, csquery, dns, dnsquery − network database

SYNOPSIS

`ndb/query` *attr value* [*rattr*]
`ndb/mkhash` *file attr*
`ndb/cs`
`ndb/csquery`
`ndb/dns` [`-s`]
`ndb/dnsquery`
`ndb/mkdb`

DESCRIPTION

The network database holds administrative information used by network programs such as *bootp*(8), *ipconfig*(8), *con*(1), etc.

Ndb/query searches the database for an attribute of type *attr* and value *value*. If *rattr* is not specified, all entries matched by the search are returned. If *rattr* is specified, the value of the first pair with attribute *rattr* of all the matched entries is returned.

Ndb/mkhash creates a hash file for all entries with attribute *attr* in database file *file*. The hash files are used by *ndb/query* and by the ndb library routines.

Ndb/cs is a server used by *dial*(2) to translate network names. It is started at boot time. It finds out what networks are configured by looking for `/net/*/clone` when it starts. It can also be told about networks by writing to `/net/cs` a message of the form:

```
add net1 net2 ....
```

Ndb/cs also sets the system name in `/dev/sysname` if it can figure it out. *Ndb/csquery* can be used to query *ndb/cs* to see how it resolves addresses. *Ndb/csquery* prompts for addresses and prints out what *ndb/cs* returns.

Ndb/dns is a server used by *ndb/cs* and by remote systems to translate Internet domain names. *Ndb/dns* is started at boot time. By default *dns* serves only requests written to `/net/dns`. Option `-s` causes the server to also answer domain requests sent to UDP port 53. Name resolution is performed by searching the local database and by querying remote servers. The server for a domain is indicated by a database entry containing both a *dom* and a *ns* attribute. For example, the entry for the Internet root is:

```
dom=
	ns=ns.nic.ddn.mil
	ns=kava.nisc.sri.com
	ns=aos.brl.mil
```

The root of a domain subtree served by the local database is indicated by an entry with an *soa* attribute. For example, the AT&T research domain is:

```
dom=research.att.com soa
	mb=ches.research.att.com
	ns=inet.research.att.com
	ns=research.research.att.com
```

Here, the *mb* entry is the mail address of the person responsible for the domain (default `postmaster`). Wildcarded domain names can also be used. For example, to specify a mail forwarder for all AT&T research systems:

```
dom=*.research.att.com
	mx=research.att.com
```

Ndb/dnsquery can be used to query *ndb/dns* to see how it resolves requests. *Ndb/dnsquery* prompts for commands of the form

domain-name request-type

where *request-type* can be *ip, mx, ns, cname, ...*

Ndb/mkdb is used in concert with *awk*(1) scripts to convert uucp systems files, IP hosts files, and Datakit configuration files into database files. It is very specific to the situation at Murray Hill.

EXAMPLES

```
% ndb/query sys helix
sys=helix dom=helix.research.att.com bootf=/mips/9powerboot
      ip=135.104.117.31 ether=080069020427
      dk=nj/astro/helix
      proto=il
% ndb/query sys helix ip
135.104.117.31
```

FILES

`/lib/ndb/local`	first database file searched
`/lib/ndb/global`	second database file searched
`/lib/ndb/local.*`	hash files for `/lib/ndb/local`
`/lib/ndb/global.*`	hash files for `/lib/ndb/global`
`/srv/cs`	service file for *ndb/cs*
`/net/cs`	where `/srv/cs` gets mounted

SEE ALSO

ndb(2) *ndb*(6)

NAME

newuser − adding a new user

SYNOPSIS

`rc /sys/lib/newuser`

DESCRIPTION

To establish a new user on Plan 9, add the user's name to `/adm/users` by running the `newuser` command on the console of the file server (see *users*(6) and *fs*(8)). Next, give the user a password using the `adduser` command on the console of the authentication server (see *auth*(8)). At this point, the user can bootstrap a terminal using the new name and password. The terminal will only get as far as running `rc`, however, as no `profile` exists for the user.

The *rc*(1) script `/sys/lib/newuser` sets up a sensible environment for a new user of Plan 9. Once the terminal is running `rc`, type

```
rc /sys/lib/newuser
```

to build the necessary directories in `/usr/$user` and create a reasonable initial profile in `/usr/$user/lib/profile`. The script then runs the profile which, as its last step, brings up *8½*(1). At this point the user's environment is established and running. (There is no need to reboot.) It may be prudent at this point to run *passwd*(1) to change the password, depending on how the initial password was chosen.

The `profile` built by `/sys/lib/newuser` looks like this:

```
bind -a $home/bin/rc /bin
bind -a $home/bin/$cputype /bin
font = /lib/font/bit/pelm/euro.9.font
switch($service){
case terminal
      prompt=('term% ' '       ')
      fn term%{ $* }
      exec 8½
case cpu
      bind -b /mnt/term/mnt/8½ /dev
      prompt=('cpu% ' ' ')
      echo -n $sysname > /dev/label
      fn cpu%{ $* }
      news
case con
      prompt=('cpu% ' ' ')
      news
}
```

Sites may make changes to `/sys/lib/newuser` that reflect the properties of the local environment.

Use the `-c` option of *mail*(1) to create a mailbox.

SEE ALSO

passwd(1), *8½*(1), *namespace*(4), *users*(6), *auth*(8), *fs*(8)

NAME

disk/prep − make disk partition table

SYNOPSIS

`disk/prep` [`-r`] *special*

DESCRIPTION

A partition table is stored on a disk to specify the division of the physical disk into a set of logical units. On Plan 9 the partition table is a list of triples: name, starting sector, and ending sector. The first two partitions must have the names `disk` and `partition`; the `disk` partition records the starting and ending sectors for the whole disk, and the `partition` partition, typically the last sector on the disk, holds the partition table itself.

Special is the maximal prefix of names of the logical units on the disk, for example `#w/hd0`. *Prep* reads and prints the associated partition table and then enters a simple interactive mode to control editing the table.

The single option `-r` (readonly) prohibits writing the table on disk.

SEE ALSO

floppy(3), *hard*(3)

NAME

qer − queue a request and associated data

SYNOPSIS

`qer` *root tag reply args*

DESCRIPTION

Qer creates a control and a data file in a queue directory. The control file contains one line with the *tag*, *reply*, and *args*. The data file contains the standard input to *qer*. The files are created in the directory *root*/*user*, where *user* is the contents of `/dev/user`. *Mktemp*(2) is used to create the actual names of the control and data file.

The *tag* is used by *runq*(8) to identify the type of request. *Reply* is a mail address to be sent error notifications when processing the request.

FILES

root/*user*	queue directory for *user*
root/*user*/`D.XXXXXX`	data file
root/*user*/`C.XXXXXX`	control file

SEE ALSO

runq(8)

NAME

runq − process all requests in a queue

SYNOPSIS

`runq` [`-ad`] *root cmd*

DESCRIPTION

Without the `-a` option, *runq* processes all requests in the directory *root*/*user*, where *user* is the contents of `/dev/user`. A request is defined as a control and a data file (see *qer*(8)). The request is processed by executing the command *cmd* with the contents of the control file as its arguments, the contents of the data file as its standard input, and standard error appended to the error file `E.XXXXXX`.

The action taken by *runq* depends on the return status of *cmd*. If *cmd* returns a null status, the processing is assumed successful and the control, data, and error files are removed. If *cmd* returns an error status containing the word `Retry`, the files are left to be reprocessed at a later time. For any other status, an error message is mailed to the requester and the files are removed.

To avoid reprocessing files too often, the following algorithm is used: a data file younger than one hour will not be processed if its error file exists and was last modified within the preceding 10 minutes. A data file older than one hour will not be processed if its error file exists and was last modified within the preceding hour.

The `-a` option forces *runq* to process all the requests under the root, not just a single user's.

The `-d` option causes debugging output on standard error describing the progress through the queues.

FILES

root/*user*	queue directory for *user*
root/*user*/`D.XXXXXX`	data file
root/*user*/`C.XXXXXX`	control file
root/*user*/`E.XXXXXX`	error file

SEE ALSO

qer(8)

NAME

scuzz − SCSI target control

SYNOPSIS

`scuzz` [`-q`] [*target-id*]

DESCRIPTION

Scuzz is an interactive program for exercising raw SCSI devices. It reads commands from standard input and applies them to a SCSI target. If *target-id* is given on the command line, an `open` (see below) is immediately applied to the target. On successful completion of a command, `ok` *n* is printed, where *n* is the number of bytes transferred to/from the target; the `-q` command line option suppresses the `ok` message.

Commands

`help` *command*

`Help` is rudimentary and prints a one line synopsis for the named *command*, or for all commands if no argument is given.

`probe` `Probe` attempts an `inquiry` command on all SCSI target ids, and prints the result preceded by the id of those targets which respond.

The `help` and `probe` commands may be given at any time.

`open` *target-id*

`Open` must be given before any of the remaining commands will be accepted. Internally, `open` issues `ready` then `inquiry`, followed by a device class-specific command to determine the logical block size of the target.

`close` `Close` need only be given if another target is to be opened in the current session.

The remaining commands are in two groups, generic SCSI commands, and those specific to the Philips CDD521 Compact Disc Recorder (`flushcache` onwards). With the exception of the `read`, `write`, `space`, and `wtrack` commands, all arguments are in the style of ANSI-C integer constants.

`ready` Test Unit Ready checks if the unit is powered up and ready to do `read` and `write` commands.

`rezero`

Rezero Unit requests that a disc be brought to a known state, usually by seeking to track zero.

`rewind`

Rewind positions a tape at the beginning of current partition (there is usually only one partition, the beginning of tape).

`reqsense`

Request Sense retrieves Sense Data concerning an error or other condition and is usually issued following the completion of a command that had check-condition status. *Scuzz* automatically issues a `reqsense` in response to a check-condition status and prints the result.

`format`

Format Unit performs a ‘‘low level’’ format of a disc.

`rblimits`

Read Block Limits reports the possible block lengths for the logical unit. Tapes only.

`read` *file nbytes*

`Read` transfers data from the target to the host. A missing *nbytes* causes the entire device to be read.

`write` *file nbytes*

Write transfers data from the host to the target. A missing *nbytes* causes the entire input file to be transferred.

The first argument to the `read`, `write`, and `wtrack` (q.v.) commands specifies a source (`write` and `wtrack`) or destination (`read`) for the I/O. The argument is either a plain file name or | followed by a command to be executed by *rc*(1). The argument may be quoted in the style of *rc*(1).

`seek` *offset whence*

Seek requests the target to seek to a position on a disc, arguments being in the style of *seek*(2); *whence* is 0 by default.

Scuzz maintains an internal notion of where the current target is positioned. The `seek`, `read`, `write`, `rewind`, `rezero`, and `wtrack` commands all manipulate the internal offset.

`filemark` *howmany*

Write Filemarks writes one (default) or more filemarks on a tape.

`space [-b] [-f] [[--]` *howmany*`]`

Space positions a tape forwards or backwards. The arguments specify logical block (`-b`) or filemark (`-f`) spacing; default is `-b`. If *howmany* is negative it specifies spacing backwards, and should be preceded by `--` to turn off any further option processing. Default is 1.

`inquiry`

Inquiry is issued to determine the device type of a particular target, and to determine some basic information about the implemented options and the product name.

`modeselect` *bytes...*

Mode Select is issued to set variable parameters in the target. *Bytes* given as arguments comprise all the data for the target; see an appropriate manual for the format.

`modesense [`*page* `[`*nbytes*`]]`

Mode Sense reports variable and fixed parameters from the target. If no *page* is given, all pages are returned. *Nbytes* specifies how many bytes should be returned.

`start` *code*

`stop` *code*

`eject` *code*

`Start`, `stop`, and `eject` are synonyms for Start/Stop Unit with different default values of *code*. Start/Stop Unit is typically used to spin up and spin down a rotating disk drive. *Code* is 0 to stop, 1 to start and 3 to eject (if the device supports ejection of the medium).

`capacity`

Read Capacity reports the number of blocks and the block size of a disc.

The remaining commands are specific to the CDD521 Compact Disc Recorder. A brief description of each is given; see the manual for details of arguments.

`flushcache`

The Flush Cache command forces data in the cache memory of the CDD521 to be written to the physical medium.

`rdiscinfo [`*track/session-number* `[`*ses*`]]`

The Read TOC/PMA command transfers data from one of the tables of contents (TOC or PMA) on the CD medium.

`fwaddr [`*track* `[`*mode* `[`*npa]*

The First Writeable Address command reports the next logical writeable address for the next `write` command.

`treserve` *nbytes*

The Reserve Track command reserves one track on the disc. Tracks can only be reserved in successive order.

`trackinfo` *track*

Read Track Info reports the starting address, the length of a given track on the disc and the number of free blocks in that track.

`wtrack`*file* [*nbytes* [*track*

Write Track sets up for track-writing if *nbytes* is 0 (default), or writes a complete track. See `write` above.

`load`

`unload`

Load and Unload open or close the tray.

`fixation` [*toc-type*]

Fixation writes table of contents (TOC) and LEADOUT information to the disc to complete a session.

FILES

`#S/`*scsi-target-id*`/cmd`	raw SCSI interface for command and status.
`#S/`*scsi-target-id*`/data`	raw SCSI interface for I/O.

SEE ALSO

scsi(3)
Small Computer System Interface - 2 (X3T9.2/86-109), Global Engineering Documents
SCSI Command Set CDD521/10, Philips IMS
SCSI Bench Reference, ENDL Publications

BUGS

Only a limited subset of SCSI commands has been implemented (as needed).

Only one target can be open at a time.

LUNs other than 0 are not supported.

No way to force 6 or 10 byte commands.

NAME

Digital Pathways SecureNet Key − remote authentication box

DESCRIPTION

The *SecureNet* box is used to authenticate connections to Plan 9 from a foreign system such as a Unix machine or plain terminal. The box, which looks like a calculator, performs DES encryption with a key held in its memory. Another copy of the key is kept on the authentication server. Each box is protected from unauthorized use by a four digit PIN.

When the system requires SecureNet authentication, it prompts with a numerical challenge. The response is compared to one generated with the key stored on the authentication server. Respond as follows:

Turn on the box and enter your PIN at the `EP` prompt, followed by the `ENT` button. Enter the challenge at `Ed` prompt, again followed `ENT`. Then type to Plan 9 the response generated by the box. If you make a mistake at any time, reset the box by pressing `ON`. The authentication server compares the response generated by the box to one computed internally. If they match, the user is accepted.

The box will lose its memory if given the wrong PIN five times in succession or if its batteries are removed.

To reprogram it, type a `4` at the `E0` prompt.

At the `E1` prompt, enter your key, which consists of eight three-digit octal numbers. While you are entering these digits, the box displays a number ranging from 1 to 8 on the left side of the display. This number corresponds to the octal number you are entering, and changes when you enter the first digit of the next number.

When you are done entering your key, press `ENT` twice.

At the `E2` prompt, enter a PIN for the box.

After you confirm the PIN at the `E3` prompt, you can use the box as normal.

You can change the PIN using the following procedure. First, turn on the box and enter your current PIN at the `EP` prompt. Press `END` three times; this will return you to the `EP` prompt. Enter your PIN again, followed by `ENT`; you should see a `Ed` prompt with a − on the right side of the display. Enter a `0` and press `ENT`. You should see the `E2` prompt; follow the instructions above for entering a PIN.

The *SecureNet* box performs the same encryption as the `netcrypt` routine (see *encrypt*(2)). The entered challenge, a decimal number between 0 and 100000, is treated as a text string with trailing binary zero fill to 8 bytes. These 8 bytes are encrypted with the DES algorithm. The first four bytes are printed on the display as hexadecimal numbers. However, when set up as described, the box does not print hexadecimal digits greater than 9. Instead, it prints a 2 for an A, B, or C, and a 3 for a D, E, or F. If a `5` rather than a `4` is entered at the `E0` print, the hexadecimal digits are printed. This is not recommended, as letters are too easily confused with digits on the *SecureNet* display.

SEE ALSO

login(8), *encrypt*(2), *auth*(2)
Digital Pathways, Mountain View, California

BUGS

The box is too clumsy. If carried in a pocket, it can to turn itself on and wear out the batteries.

NAME

snoopy − spy on Ethernet packets

SYNOPSIS

`snoopy [ -abeiltup9 ][ -np ]`

DESCRIPTION

Snoopy displays the header and first 20 data bytes of packets received from the local Ethernet. The packets displayed depend on the options chosen. The following options each select packets from a particular protocol. If more than one flag is given, packets from all those protocols are displayed.

`a`	ARP
`b`	BOOTP
`e`	all Ethernet packets
`i`	IP
`l`	IL
`t`	TCP
`u`	UDP

By default all addresses are translated into system names. The option `n` suppresses this.

Snoopy runs in promiscuous mode by default, displaying all packets it can capture from the Ethernet. The option `p` causes only packets sent to or from the system *snoopy* is running on to be displayed.

Option `9` causes the data of TCP and IL messages to be interpreted and displayed as 9P messages.

FILES

`/net/ether`
Ethernet device

BUGS

The CPU servers do not take well to running in promiscuous mode. If run on them, *snoopy* may kill their Ethernets.

NAME

swap − establish a swap file

SYNOPSIS

`swap` *file*

DESCRIPTION

Swap establishes a file or device for the system to swap on. If *file* is a device, the device is used directly; if a directory, a unique file is created in that directory on which to swap. The environment variable `swap` is set to the full name of the resulting file. The number of blocks available in the file or device must be at least the number of swap blocks configured at system boot time.

If a swap channel has already been set and no blocks are currently valid in the file the old file will be closed and then replaced. If any blocks are valid on the device an error is returned instead.

NAME

sysmon, stats − display graphs of system activity

SYNOPSIS

`sysmon` [*machine*]

`stats`

DESCRIPTION

Sysmon displays a rolling graph of various statistics collected by the operating system. The statistics may be from a remote machine. A sample value is taken once per second. The number in the top left corner of the graph gives the peak value for the duration of the graph.

Lines across the graph represent 75%, 50% and 25% of the peak value. Clicking the mouse buttons anywhere in the window selects a new parameter to monitor. The parameters are:

`mem`	total pages of active memory. The memory is displayed as a fraction of the machine's total memory.
`ether`	number of packets sent and received per second.
`swap`	number of valid pages on the swap device. The swap is displayed as a fraction of the number of swap pages configured by the machine.
`contxt`	number of process context switches per second.
`intr`	number of interrupts per second.
`fault`	number of memory faults per second.
`tlbmiss`	number translation lookaside buffer misses per second.
`tlbpurge`	number translation lookaside buffer flushes per second.
`load`	system load average. The load is computed as a running average of the number of processes ready to run multiplied by 1000 to give some precision.

The *stats* program is invoked by *sysmon* to display the graph.

FILES

`/net/*/[0-n]/stats`
`#c/sysstat`

BUGS

Some machines do not have TLB hardware.

NAME

aux/vga − setup VGA card

SYNOPSIS

`aux/vga` [`-c` *chip*] [`-f` *configfile*] [`-t` *vgatype*] [`-i`] [`-D`] [*xsize* [*ysize* [*zsize*]]]

DESCRIPTION

`Aux/vga` configures VGA cards and the kernel for various display sizes and depths. It scans a *configfile* (usually `/lib/vgadb`) for a given *vgatype* and size. If no *vgatype* is specified, the environment variable `vgatype` is used. The record name typically represents a monitor/SVGA card pair, like `tseng-nec4`.

A good super VGA card and high quality monitor will support 320x200, 640x480, 800x600, 1024x768, and possibly 1280x1024. All usually support 1, 2, and 4-bit pixel depths with their 4-bit planar modes. All but the last support 8-bit pixels as well. For 1-bit displays, only the *xsize* is needed.

The *configfile* contains a series of records. Each record has a name starting in column one followed by indented configuration information: the screen x, y, and z dimensions, a series of standard VGA register values, a quoted string giving the VGA chip type, and possibly some more special registers values for that chip type.

We currently support the Tseng Labs T4000 chip on a Cardinal 765 VGA board, the Paradise PVGA1A chip on the AT&T VGA-600 board, limited support for the Trident 8900, and some values for the Safari and Nomad computers.

When debugging new configurations, the following switches and commands are useful:

`-c`*chip*	Set the chip type. This is usually set by the *vgatype* record. The default chip type is read from `#v/vgatype`.
`-D`	Read the registers back after a new configuration is loaded and print the differences.
`-i`	Interactive mode. It accepts the following commands:

`d`	Read and display VGA registers.
`q, x`	Quit program.
`arr`	Display attribute register *rr*.
`crr`	Display crt register *rr*.
`grr`	Display graphics register *rr*.
`srr`	Display sequence register *rr*.
`Arr=hh`	Set attribute register hex *rr* to hex value *hh*.
`Crr=hh`	Set crt register hex *rr* to hex value *hh*.
`Grr=hh`	Set graphics register hex *rr* to hex value *hh*.
`Srr=hh`	Set sequence register hex *rr* to hex value *hh*.

EXAMPLES

```
vgatype=generic aux/vga 640
```

Set a generic VGA to a standard size.

```
vgatype=tseng-nec5 aux/vga 1024 768 2
```

Setup a nice 2-bit grey scale display on a NEC 5 with Tseng Lab's chip.

FILES

`/lib/vgadb`	VGA configuration file.

SEE ALSO

vga(3)

BUGS

It takes a lot of documentation, or a lot of work to support a new VGA chip. PC color displays don't approach black-and-white workstation clarity and size unless they are *very* expensive.

NAME

intro − introduction to raster image software

DESCRIPTION

Plan 9 provides a suite of commands and library routines to create and manipulate files containing gray-scale and full-color images. Section 9 of the manual is divided into subsections numbered like the main manual sections: 9.1 for commands, 9.2 for library routines, 9.6 for file formats.

Picture files are two-dimensional arrays of multi-byte records with a textual header describing the dimensions of the image, the algorithm used to encode the file, and whatever other information programs may wish to preserve. *picfile*(9.6) describes the file format; *picopen*(9.2) describes a library of routines to read and write picture files.

`/bin/fb` contains a collection of programs to manipulate picture files.

Drop displays a picture file in an *8½*(1) window or on a raw Plan 9 terminal. *Examine* similarly displays an image and allows interactive examination of its pixel values. *Picinfo* displays the header of a picture file on its standard output. *Pcp* copies picture files, modifying header attributes as requested and updating the encoded picture array correspondingly. It can clip a subwindow out of a picture, permute, delete and rename channels, change the encoding type and even convert full-color images to monochrome and vice-versa. *Hed* is a more brute-force version of *pcp* that can apply an arbitrary *sed*(1) script to a picfile header. It copies the image array verbatim and can thus convert precious images into garbage or vice-versa.

Dumppic, gif2pic, picopic and *Face2pic* convert files in various alien formats to *picfile*(9.6) format. *Pic2ps* converts picfiles to encapsulated Postscript. *Nohed* removes the header from a picture file. When applied to a `TYPE=dump` picture this converts it into the ubiquitous ‘raw dump’ format. *Mugs* is an interactive program to convert picfiles into 48×48 icons of the sort used by `seemail` (see *mail*(1)).

Some commands create simple images out of whole cloth. *Card* writes an image of constant color. *Ramp* creates an image that is one color at one edge and changes linearly to another color at the opposite edge.

Twb and its subroutine *dpic* convert *troff*(1) input into anti-aliased images.

Aplot reads a square array of data points and draws an anti-aliased perspective plot of the surface it defines.

There are numerous commands that read one or more images and write a modified image on standard output. See *remap*(9.1), *filters*(9.1), *floyd*(9.1), *he*(9.1), *lam*(9.1), *lerp*(9.1), *logo*(9.1), *lum*(9.1), *quantize*(9.1), *resample*(9.1) *transpose*(9.1) and *xpand*(9.1) for descriptions.

Moto is an animator’s command language. It converts concise descriptions of simultaneous processes overlapping in time into sequential command files suitable for producing frames of an animation.

SEE ALSO

Sections *add*(2), *balloc*(2), *cachechars*(2), *subfalloc*(2), *bitblt*(2), *event*(2), *frame*(2), *print*(2), *bit*(3), *layer*(2), *bitmap*(6) and *font*(6) describe the standard Plan 9 interactive bitmap graphics interface.

NAME

aplot − isometric plots of data arrays

SYNOPSIS

`fb/aplot` [`-a`] [`-l`*lightfile*] [`-t`*type*] [`-r`*range*] [`-w`*x0 y0 x1 y1*] *file*

DESCRIPTION

Aplot draws an anti-aliased isometric perspective plot of the square array of elevations that it reads from *file.* The output is a picture file, written on standard output.

Option `-t` specifies the type of the data in the binary file. Possible *types* are

`s`	`short`
`i`	`int`
`l`	`long`
`f`	`float`
`d`	`double`
`c`	`char`
`u`	`unsigned char`

The default is `-t f`.

Option `-w` sets the `WINDOW=` attribute of the output image. By default, the image is drawn in a 640×512 window.

Normally, the data is scaled to make the plot fill the window. This default scaling can be overridden by option `-r`, in which case the data is scaled so that *range* is the magnitude of data values that would make a plot that just fills the window vertically.

Option `-l` gives the name of a file describing how to shade the surface and how shiny the surface is. By default, a not-at-all shiny surface is lit from above by a single light source. The *lightfile* contains lines of the following forms:

`light` *x y z brightness*
specifies light source of the given brightness shining in direction *(x,y,z).* There can be up to 16 light sources. The default light is in direction (2,3,9) and has brightness 1.

`ambient` *brightness*
specifies the brightness of then ambient (non-directional) light. The default is 0.02.

`diff` *reflectance*
sets the amount of diffuse reflection from the surface. The default is 0.98.

`spec` *reflectance*
sets the amount of specular reflection from the surface. The default is 0.

`bump` *height*
sets the width of the specular reflection bump. Larger numbers produce tighter (less diffuse) bumps. The default is 80.

Option `-a` suppresses writing an *alpha* channel into the output file. By default, the output has `CHAN=ma`.

SEE ALSO

picfile(9.6), *filters*(9.1)

BUGS

Input files assumed to use native byte order and floating point format, and so are not transportable, except for `-t u`.

NAME

card, ramp − create simple color fields

SYNOPSIS

`fb/card` [`-c`*rgba*] [`-w`*x0 y0 x1 y1*] *red* [*green blue*] [*alpha*]

`fb/ramp` [`-v`] [`-w`*x0 y0 x1 y1*] [[*leftcolor*] *rightcolor*]

DESCRIPTION

Card writes a constant (all pixels equal) picture on standard output. The *red, green blue* and *alpha* arguments are numbers between 0 and 255. *Green* and *blue* default equal to *red. Alpha* defaults to 255. Option `-c` specifies the `CHAN=` attribute of the picture file. The default is `CHAN=m`, `CHAN=ma`, `CHAN=rgb`, or `CHAN=rgba` depending on which of *red, green blue* and *alpha* are specified.

Ramp creates a picture file whose pixel values range from *leftcolor* to *rightcolor* across each scan line, writing on standard output. *Leftcolor* defaults to 0; *rightcolor* defaults to 255. If three arguments are given for *leftcolor* and *rightcolor* the output will have `CHAN=rgb`. Otherwise, it will have `CHAN=m`.

Option `-v` causes *ramp* to make a vertical ramp (*leftcolor* at the top, *rightcolor* at the bottom).

For both commands, option `-w` specifies the size of the picture. The default gives `WINDOW=0 0 1280 1024`.

SEE ALSO

picfile(9.6)

NAME

cmap − color map format

DESCRIPTION

A color map is a 256×3-byte translation table for color values. In a monochrome picture, pixel values index the color map to yield red, green and blue, like this:

```
unsigned char cmap[256][3];
red=cmap[pixel][0];
green=cmap[pixel][1];
blue=cmap[pixel][2];
```

In a full-color picture, the color map is, in effect, three intensity-compensation tables: `cmap[red][0]` maps red channels, `cmap[green][1]` maps green channels and `cmap[blue][2]` maps blue channels.

A colormap file is just a 768-byte file containing the color map, stored in the order implied by the declaration of `cmap` above.

NAME

drop, save, flip − copy picture files to and from screen

SYNOPSIS

`fb/drop` [*input*]

`fb/save`

`fb/flip` [`-r` *fps*] [`-p`] p1 p2 ...

DESCRIPTION

Drop displays its argument picture file (default standard input) in the middle of a terminal screen or `8½` window. As most Plan 9 terminals are grey-scale devices with only a few bits per pixel, it computes the luminance of color images and uses error-diffusion dither to quantize the pixel values. *Save* writes a picfile containing its window (or screen if `8½` is not running) onto its standard output.

Flip displays many picfiles in sequence in a loop. The pictures must be the same size, and must fit in memory. The pictures are all loaded into main memory and then sent to the display as required using `wrbitmap` (see *balloc*(2)), so the machine running *flip* can be remote; a CPU server can be used if there are many large frames. The `-r` option sets the display rate in frames per second. By default *flip* displays as fast as it can: about 15 frames per second for a small picture on a Magnum. The `-p` flag causes a one-second pause at the end of the loop.

SEE ALSO

picfile(9.6)

NAME

dumppic, face2pic, gif2pic, nasa2pic, pcx2pic, picopic, utah2pic − convert other formats to picture files

SYNOPSIS

`fb/dumppic` *input xsize ysize channels*

`fb/face2pic` [*facefile*]

`fb/gif2pic` *file*

`fb/nasa2pic` *file*

`fb/pcx2pic` [`-r`] [*file*]

`fb/picopic` *red green blue xsize ysize*

`fb/utah2pic` *file*

DESCRIPTION

Dumppic copies *input* to its standard output, adding a `TYPE=dump` picture file header. *Xsize* and *ysize* are the width and height of the picture. *Channels* is the value of the output's `CHAN=` attribute. *Face2pic* reads *facefile* (default standard input), a file in the Usenix face-saver format, and converts it to *picfile*(9.6) format, writing the result on standard output.

Picopic creates a `TYPE=pico` image from the files *red, green* and *blue* which must be raw (headerless) dumps in scanline order of size *xsize*×*ysize.*

Gif2pic reads a CompuServ GIF format picture from *input* and converts it to *picfile*(9.6) format, written on standard output.

Nasa2pic reads a NASA satellite image and converts it to *picfile*(9.6) format, written on standard output. It can decipher only images with a single 8-bit channel. NASA images typically have copious annotations in their headers; these are mostly lost.

Utah2pic reads a Utah format image and converts it to *picfile*(9.6) format, written on standard output.

Pcx2pic reads a Paintbrush PCX format picture from *input* (or standard input) and converts it to *picfile*(9.6) format, written on standard output. `TYPE=dump` is generated by default; the `-r` flag selects `TYPE=runcode`.

SEE ALSO

picfile(9.6)

NAME

examine − examine pixel values interactively

SYNOPSIS

`fb/examine` [*input*]

DESCRIPTION

Examine displays an approximation of its argument picture file (default standard input) in the middle of the screen or window and waits for mouse input. Button 1 selects pixels, whose coordinates and values are written to standard output. Button 2 refreshes the display, which can be overwritten by *examine*'s output. Button 3 exits.

SEE ALSO

picfile(9.6)

BUGS

For compatibility with other programs, buttons 2 and 3 should pop up menus. But single-item menus are silly, and it's too much to have two of them.

NAME

adapt, ahe, crispen, laplace, edge, edge2, edge3, extremum, median, nonoise, smooth, shadepic − image neighborhood operators

SYNOPSIS

`fb/adapt` [*input*]

`fb/ahe` [*input*]

`fb/crispen` [*input*]

`fb/laplace` [*input*]

`fb/edge` [*input*]

`fb/edge2` [*input*]

`fb/edge3` [*input*]

`fb/extremum` [*input*]

`fb/median` [*input*]

`fb/nonoise` [*input*]

`fb/smooth` [*input*]

`fb/shadepic` [`-l`*x y z*] [*input*]

DESCRIPTION

Gathered here are descriptions of programs that compute the pixels of an output image by performing some operation on a neighborhood of each pixel of their input image (default standard input). Each program writes the output image on standard output. The programs process multi-channel inputs by treating each channel independently.

Adapt performs adaptive contrast enhancement by examining the 7×7 region centered on each input pixel, remapping the center pixel linearly in a way that would send the neighborhood's maximum value to 255 and its minimum to 0. To avoid divide checks, no mapping is done if all pixels in the region have the same value.

Ahe performs adaptive histogram equalization by examining the 17×17 region centered on each input pixel, counting the number of pixels whose value is less than the center pixel. (It counts ½ for each pixel equal to the center value.) Output pixel values are 255 times the count divided by the window size.

Crispen examines the 3×3 region surrounding each input pixel, computing 9 times the center pixel minus the sum of its eight neighbors. This is a fairly extreme high-pass filter and sharpens edges substantially.

Laplace computes 5 times the center pixel minus the sum of its four vertical and horizontal neighbors. This adds a 3×3 discrete Laplacian to the original image, and is a less extreme high-pass filter than *crispen.*

Edge, edge2 and *edge3* detect edges in various ways. *Edge* examines the 3×3 region surrounding each input pixel, outputting 8 times the center value minus the sum of its eight neighbors.

Edge2 applies a Sobel operator to the input image. It approximates the image's gradient by finite differences on a 3×3 neighborhood, outputting the vector length of the gradient approximation.

Edge3 likewise approximates the gradient of the input image. The output is roughly the phase angle of the gradient approximation, scaled between 0 and 255.

Extremum examines the 3×3 region surrounding each input pixel, outputting the value that differs most from the center value. In case of a tie, the larger candidate is chosen.

Median does noise reduction by replacing each pixel of the input image by the median of the 3×3 region surrounding it.

Nonoise implements the Bayer-Powell noise reduction filter. It computes the average value of the eight neighbors of each pixel of the input image, and substitutes it for the pixel value if the two differ by more than 64.

Smooth low-pass filters its input image by convolution with a Bartlett window.

Shadepic treats its input image as an array of elevations. At each pixel it approximates the normal vector to the height-field by finite differences on a 3×3 neighborhood and outputs 255 times its dot product with the unit vector in the light-source direction specified by option `-l` (default 1,-1,1). If the dot product is negative, it is clamped at zero. (This computation is just Lambertian diffuse reflection.)

SEE ALSO

picfile(9.6)

BUGS

There are too many weird wired-in sizes, like 17× 17 and 7×7.

NAME

floyd, halftone, hysteresis − create 1-bit images by dithering

SYNOPSIS

`fb/floyd` [*input*]

`fb/halftone` *screen* [*input*]

`fb/hysteresis` *low high* [*input*]

DESCRIPTION

Floyd reads a grey-scale input file (default standard input), and reduces it to one bit per pixel using Floyd-Steinberg error-diffusion dither, as improved by Ulichney. The resulting `TYPE=bitmap` picture file is written to standard output.

Ulichney's algorithm involves randomly varying the Floyd-Steinberg diffusion coefficients. As the random number generator is seeded from the clock, *floyd* may produce different output if rerun on the same input.

Halftone reduces grey-scale images to one bit per pixel using ordered dither. The *screen* argument is the name of a file containing a dither matrix. *Halftone* searches for screens in `/lib/fb/screens`.

Hysteresis creates one-bit-per-pixel images by thresholding with hysteresis. Any value in the input image less than *low* is mapped to zero. Any input value less than *high* is mapped to zero if any of its eight neighbors is less than *low*. If *low* and *high* are equal, this is just an ordinary thresholding operation.

Hysteresis makes a useful edge-detection operator if used on a high-pass filtered image.

SEE ALSO

picfile(9.6)

FILES

`/lib/fb/screens/*`

NAME

getcmap − read a color map from a file

SYNOPSIS

```
#include <libg.h>

#include <fb.h>

int getcmap(char *name, uchar *map)
```

DESCRIPTION

Getcmap retrieves the named colormap and stores it in *map.* Usually *name* is the name of a colormap file (see *cmap*(9.6)) or a picfile with a `CMAP=` attribute. If the file is not found, it is sought in `/lib/fb/cmap`.

If the *file* cannot be found, and its name has the form `gamma`*number,* a colormap is fabricated with all three channels of its *n*th entry set to $255\times(n/255)^{1/number}$. If the name is just `gamma`, *number*=2.3 is assumed.

There is no *putcmap*, because `write` (in *read*(2)) can do the job.

SEE ALSO

cmap(9.6), *picfile*(9.6) *rgbpix*(2)

NAME

getflags, usage − process flag arguments in argv

SYNOPSIS

```
#include <libg.h>

#include <fb.h>

int getflags(int argc, char *argv[], char

int usage(char *tail)

extern char **flag[], cmdline[], *cmdname, *flagset[];
```

DESCRIPTION

Getflags digests an argument vector *argv*, finding flag arguments listed in *flags*. *Flags* is a string of flag letters. A letter followed by a colon and a number is expected to have the given number of parameters. A flag argument starts with '−' and is followed by any number of flag letters. A flag with one or more parameters must be the last flag in an argument. If any characters follow it, they are the flag's first parameter. Otherwise the following argument is the first parameter. Subsequent parameters are taken from subsequent arguments.

The global array *flag* is set to point to an array of parameters for each flag found. Thus, if flag `-x` was seen, `flag['x']` is non-zero, and `flag['x'][i]` is the flag's *i*th parameter. If flag `-x` has no parameters `flag['x']==flagset`. Flags not found are marked with a zero. Flags and their parameters are deleted from *argv*. *Getflags* returns the adjusted argument count.

Getflags stops scanning for flags upon encountering a non-flag argument, or the argument `--`, which is deleted.

Getflags places a pointer to *argv*`[0]` in the external variable *cmdname.* It also concatenates the original members of *argv*, separated by spaces, and places the result in the external array *cmdline*.

Usage constructs a usage message, prints it on the standard error file, and exits with status 1. The command name printed is *argv*`[0]`. Appropriate flag usage syntax is generated from *flags*. As an aid, explanatory information about flag parameters may be included in *flags* in square brackets as in the example. *Tail* is printed at the end of the message. If *getflags* encountered an error, *usage* tries to indicate the cause.

EXAMPLES

```
main(int argc, char *argv[]){
      if((argc=getflags(argc, argv, "vinclbhse:1[expr]", 1))==-1)
            usage("[file ...]");
}
```

might print:

```
Illegal flag -u
Usage: grep [-vinclbhs] [-e expr] [file ...]
```

SEE ALSO

ARG(2)

DIAGNOSTICS

Getflags returns −1 on error: a syntax error in *flags,* setting a flag more than once, setting a flag not mentioned in *flags*, or running out of argv while collecting a flag's parameters.

NAME

he − histogram equalization

SYNOPSIS

`fb/he` [*input*]

DESCRIPTION

He reads a picture file (default standard input) and maps its pixel values monotonically to make their histogram as even as possible. The resulting picture file is written to standard output.

SEE ALSO

picfile(9.6)

NAME

hed, nohed − edit or remove picture file header

SYNOPSIS

`fb/hed` [`-n`] *script input*

`fb/nohed` *input*

DESCRIPTION

Hed runs *sed*(1) with the given *script* on the input picture file's header. The resulting picture file is written on standard output.

Nohed removes the *picfile*(9.6) header from the input image. This is useful mostly for turning a `TYPE=dump` image into a raw bitmap dump.

SEE ALSO

sed(1), *picfile*(9.6)

NAME

lam, posit, piccat, picjoin − combine several images

SYNOPSIS

`fb/lam` *input ...*

`fb/posit` *input ...*

`fb/piccat` *input ...*

`fb/picjoin` *input ...*

DESCRIPTION

Lam overlays (‘‘laminates’’) several picture files, writing the resulting picture file to standard output. The output `WINDOW=` attribute is the smallest rectangle that contains all of the input rectangles. Each pixel of the output image takes its value from the last-mentioned input image that covers that pixel. Output pixels not covered by any input image are set to zero.

Posit performs similarly, except that output pixels are computed by compositing the corresponding input pixels, with later input pictures `over` earlier ones. If the input images have no *alpha* channel, *posit* has the same effect as *lam.*

Piccat concatenates a list of picture files, each above the next, writing the result to standard output. The width of the output file will be that of the widest input picture. If any of the input pictures are more narrow than that, the space to their right will be zero in the output picture.

Picjoin is similar, but joins the pictures left-to right.

SEE ALSO

picfile(9.6), Thomas Porter and Tom Duff, ‘‘Compositing Digital Images,’’ *Computer Graphics,* Vol 18, No. 3 (1984), pp. 253-259

BUGS

All pictures must have identical `CHAN=` attributes.

NAME

lerp − linear combinations of images

SYNOPSIS

`fb/lerp` *file fraction* ... [*file*]

DESCRIPTION

Lerp computes a linear combination of a number of input images. Each input file name is followed by a floating-point fraction by which to scale its pixel values. The fraction after the last image may be omitted, in which case one minus the sum of the other fractions is used. The result image is written to standard output.

Nothing prevents the fractions from being smaller than zero or larger than one. Output pixel values that fall below zero or above 255 are clamped.

SEE ALSO

picfile(9.6)

NAME

logo − convert image into an AT&T logo

SYNOPSIS

`fb/logo` [`-dl`] *height* [*file*]

DESCRIPTION

Logo converts the input image into a simulacrum of the AT&T logo. *Height* is the height in pixels of the logo's stripes. The result image is written to standard output.

Option `-d` makes the background dark (the default); `-l` makes the background light.

SEE ALSO

picfile(9.6)

NAME

lum − compute luminance

SYNOPSIS

`fb/lum` [*input*]

DESCRIPTION

Lum computes the luminance of the input picture using the NTSC formula *L=.299R+.587G+.114B.* Pixel values are mapped through the input image's color map, if any. The resulting image is written to standard output.

SEE ALSO

picfile(9.6)

NAME

moto – create animation scripts

SYNOPSIS

`fb/moto` [`-f`*start end*] [`-s`*skip*] [*file* [*arg* ...]]

DESCRIPTION

Moto is a command generator tailored for an animator's needs. Its input is a concise description of the animation to be produced; its output is a command file suitable for input to *rc* or some other command interpreter. Its arguments are an optional file name containing a *moto* program (default standard input) and list of numeric parameters that are made available to the program.

A *moto* program consists of a list of groups of commands guarded by a range of frames. Groups may contain parameter ranges enclosed in brackets `[]`. For each frame, *moto* checks each group and processes those whose guards include the current frame number:

```
1,6:  clr 128
1,4:  clr -w [0,30] [0,30] [100,130] [100,130]
3,6:  clr -w [100,70] [100,70] [130,100] [130,100] 255
```

This generates

```
clr 128
clr -w 0 0 100 100
clr 128
clr -w 10 10 110 110
clr 128
clr -w 20 20 120 120
clr -w 100 100 130 130 255
clr 128
clr -w 30 30 130 130
clr -w 90 90 120 120 255
clr 128
clr -w 80 80 110 110 255
clr 128
clr -w 70 70 100 100 255
```

Two special guards, `BEGIN` and `END`, specify actions to be taken before and after processing frames. *Moto* allows complex computations inside parameter brackets:

```
1,10: clr [127.5*(1-cos([0,360]))]
```

This generates

```
clr 0
clr 29.82933350233
clr 105.35985734747
clr 191.25
clr 247.3108091502
clr 247.3108091502
clr 191.25
clr 105.35985734747
clr 29.82933350233
clr 0
```

Expressions may include constants and variables. All values are double-precision floating point numbers. The operators =, `/`, +, − (both unary and binary), <, >, <=, >=, ==, `!=`, `?:` and `!`, all with their meanings as in C, except that all results are coerced to `double`. The result of `a%b` is `a-b*(int)(a/b)`. The result of `a&&b` is `a?b:a`. The result of `a||b` is `a?a:b`. The exponentiation operator is `^`, also written `**`. The expression `[a,b]` varies from `a` to `b`, linearly as the frame number varies between the guards of the group containing the expression. The expression `a[b,c]` has the value `a*b+(1-a)*c`. Its value varies from `b` to `c` as `a` varies from 0 to 1. The expression `$`*i* has the value of the *i*'th parameter following the file name on *moto*'s command line.

The precedence of operators is, from lowest to highest:

```
=
? :
||
&&
< <= == != > >=
+ -
* / %
[ ]
^ **
- (unary) ! $
```

Expressions may be parenthesized to alter precedence.

The following math functions are available:

```
fabs floor ceil sqrt hypot sin cos tan gamma
asin acos atan exp log log10 sinh cosh tanh
```

All math functions are as described in the C library, except that angles are measured in degrees rather than radians for the trig and inverse trig functions. In addition `hypot` may have two or three arguments, `atan` may take two arguments instead of one, and may also be spelled `atan2`.

For parameterization, and to allow even more complex computations, *moto* has variables, assignment and computation groups. A computation group causes no output; rather its body is a group of expressions to be evaluated for their side effects. It is distinguished from a command group by having a double colon separating the guard and body:

```
BEGIN::     n=5
1,n:: x=512*sin([0,90])
1,n:  pcp -w 0 0 [x] 488 pic.[1,n] %0
```

This generates

```
pcp -w 0 0 0 488 pic.1 %0
pcp -w 0 0 195.93391737093 488 pic.2 %0
pcp -w 0 0 362.03867196751 488 pic.3 %0
pcp -w 0 0 473.02632064578 488 pic.4 %0
pcp -w 0 0 512 488 pic.5 %0
```

Upon occasion it is useful to split *moto*'s output into several files, under program control. A group that is separated from its guards by an at-sign `@` instead of a colon names a file into which subsequent output is to be written. For example,

```
1,5@  file.[1,5]
1,5:  This is file.[1,5].
```

creates 5 files, with names `file.1`, ..., `file.5`. Each file's contents will announce its name.

As is true for all sufficiently large programs, *moto* has a shell escape. A group separated from its guards by an exclamation point `!` instead of a colon has its result text interpreted by a subshell.

NAME

mugs − make face icons from pictures

SYNOPSIS

`mugs` [`-a`] [`-1`] [`-2`] [*file*]

DESCRIPTION

Mugs interactively converts grey-scale images in the form of *picfile*(9.6) into 48×48 icons. It is designed to run in a pipe, reading the picture from standard input unless a single *file* is given on the call. *Mugs* displays a large approximation to the original picture and a matrix of 48×48 icons of varying contrast and brightness. Button 1 selects one of the icons. Button 2 offers the menu entries:

`in` Zoom in to a finer contrast/brightness range around the selected icon. Repeated `ins` will zoom in farther.

`out` Opposite of `in`.

`reset` Set the brightness/contrast range to the maximum.

Both `in` and `out` preserve the brightness/contrast values in the selected icon. Button 3 presents a menu with entries:

`window`
Select a square window in the large picture using button 3. Touch down at the top and center of the square and slide around to adjust its size. Appropriately cropped icons will be displayed.

`depth` Toggle between 1- and 2-bit deep icons.

`write` Write the selected icon to standard output. Each write produces 48 lines of text suitable for initializing an array in C. 1-bit deep icons produce three shorts per line; 2-bit depths are written as three longs per line.

`abort` Terminate *mugs* with a non-blank error return.

`finish`
Terminate with a null status return.

Option `-a` indicates that picture files have non-square pixels with aspect ratio 1.25, as produced by some frame grabbers. Normally pixels are assumed to be square. `-1` and `-2` select the initial depth of the icons. `-2` is default.

SEE ALSO

picfile(9.6)

BUGS

Preservation of the selected icon through an `out` operation leads to strained ranges.

NAME

pcp − copy pictures

SYNOPSIS

`fb/pcp` [`-w`*x0 y0 x1 y1*] [`-o`*x y*] [`-t`*type*] [`-c`*channels*] [`-C`*channels*] [*input* [*output*]]

DESCRIPTION

Pcp copies the input picture (default standard input) to the output file (default standard output). Options control the attributes and content of the output picture.

`-w` *x0 y0 x1 y1*
causes only the given window of the input picture to be copied. By default the whole picture is copied.

`-o` *x y* causes the output picture's `WINDOW=` attribute to be translated by adding (x,y) to the input window coordinates.

`-t` *type* sets the `TYPE=` attribute of the output file. The default is to use the input file's type.

`-c` *channels*
causes only the given channels of the input picture to be copied. The default is to copy all channels. If channels not present in the input picture are specified, they are computed in the ‘‘most plausible’’ way. For example, a missing *alpha* channel is set to 255, and an `m` channel will be synthesized from `rgb` channels by computing NTSC luminance. A `0` in *channels* causes a zero channel to be written.

`-C` *channels*
sets the output `CHAN=` attribute. *Channels* must be the same length as the selected channels of the input picture.

SEE ALSO

picfile(9.6)

NAME

pic2ps − convert picture file into postscript language

SYNOPSIS

`fb/pic2ps` [`-h` *height*] [*input*]

DESCRIPTION

Pic2ps converts its input image (default standard input) into encapsulated Postscript, writing the result to standard output. If the input image is full-color, its luminance is computed first. Option `-h` sets the output image height in inches. The default height is 3", to match the default height of the `.BP` macro in `troff -mpictures`.

SEE ALSO

picfile(9.6), `/sys/lib/tmac/tmac.pictures`, the troff Postscript inclusion macros.

NAME

picfile − raster graphic image format

DESCRIPTION

Files in this format store images represented as two-dimensional arrays of multiple-channel pixels. A *picfile* consists of an textual header followed by binary data encoding the pixels in row-major order. The header is a list of attribute/value pairs separated by newlines, terminated by an empty line. Each header line has the form *name=value*. The name may not contain an ASCII NUL, newline or =; the value may not contain NUL or newline. The last line of a header is empty.

The standard attributes are described below; all but `TYPE` and `WINDOW` are optional. `TYPE` must come first; otherwise order is irrelevant. As any unrecognized attribute is passed over uninterpreted by all standard software, applications are welcome to include arbitrary annotations, like `SHOESIZE=10`, if they wish.

`TYPE=`*type*

How the pixels are encoded. Standard types are

`runcode`	A run-length encoding. The data are a sequence of (*nchan*+1)-byte records each containing a count *k* and *nchan* bytes giving a pixel value to be repeated *k*+1 times. A run may not span scanlines.
`dump`	A two-dimensional array of *nchan*-byte records in row major order.
`bitmap`	One-bit pixels, packed into bytes high bit leftmost. Zero bits are white, one bits are black. Rows are padded with zeros to a multiple of 16 bits.
`ccitt-g4`	A black-and-white image under CCITT FAX Group 4 compression. This format is highly compressive on images of text and line art. Similarly, `ccitt-g31` and `ccitt-g32` for Group 3, 1-D and 2-D.
`pico`	A sequence of *nchan* two-dimensional arrays of single bytes.
`ccir601`	Pixels are in dump order, 2 bytes per pixel encoded according to the IEEE digital component video standard.

`WINDOW=`*x0 y0 x1 y1*

The *x,y* coordinates of the upper left corner and the point just diagonally outside the lower right corner, *x* increasing to the right, *y* down.

`NCHAN=`*nchan*

The number of channels, default 1.

`CHAN=`*channels*

The names of the channels. *Channels* should be *nchan* characters long. Certain substrings of *channels* are conventionally understood by most programs that read and write picture files: `m` is a monochrome image channel, `rgb` is a full-color image, `a` is an alpha channel, and `z...` is a floating point (four-byte, single precision) *z* value. Some very old monochrome pictures have `CHAN=r`. This usage is deprecated but still recognized by some programs.

`RES=`*x y*

The digitizing resolution horizontally and vertically, in pixels/inch.

`CMAP=` (The value is empty.) A color map, a 256×3-byte translation table for color values, follows the header. In a full-color picture, each color-map row maps pixel values of the corresponding channel. In a monochrome picture, pixel values index the color map to yield red, green and blue, like this:

```
unsigned char cmap[256][3];
red=cmap[pixel][0];
green=cmap[pixel][1];
blue=cmap[pixel][2];
```

EXAMPLES

`sed '/^$/q' image`

Print a header. A sample header follows.

```
TYPE=dump
WINDOW=0 0 512 512
```

```
NCHAN=1
CHAN=m
RES=300 300
CMAP=
COMMAND= antiquantize 'halftone CLASSIC' 512.halftone LIBERTY.anticlassic
COMMAND=  halftone CLASSIC 512.liberty 512.halftone 1.75 512.halftone
COMMAND=   transpose IN OUT
COMMAND=    resample 512 IN OUT
COMMAND=     transpose IN OUT
COMMAND=      resample 512 IN OUT
COMMAND=       clip 400 400 LIBERTY OUT
```

SEE ALSO

bitmap(6)

T. Duff, 'The 10th Edition Raster Graphics System', UNIX Research System Papers, Tenth Edition

NAME

picinfo − print information about picture files

SYNOPSIS

`fb/picinfo` *file* ...

DESCRIPTION

Picinfo prints the header information as described in *picfile*(9.6) from the named *files.* If the file has no `picfile` header, it tries to guess from its size what sort of image it might be.

SEE ALSO

picfile(9.6)

NAME

picopen_r, picopen_w, picread, picwrite, picclose, rdpicfile, wrpicfile, picputprop, picgetprop, picunpack, picpack, picerror − picture file I/O

SYNOPSIS

```
#include <libg.h>

#include <fb.h>

PICFILE *picopen_r(char *name)

PICFILE *picopen_w(char *name, char *type, int x0, int y0, int w, int h,
char *chan, char *argv[], char *cmap)

int picread(PICFILE *pf, char *buf)

int picwrite(PICFILE *pf, char *buf)

void picclose(PICFILE *pf)

Bitmap *rdpicfile(PICFILE *pf, int ldepth)

int wrpicfile(PICFILE *pf, Bitmap *b)

PICFILE *picputprop(PICFILE *pf, char *name, char *value)

char *picgetprop(PICFILE *pf, char *name)

void picunpack(PICFILE *pf, char *pix, char *fmt, ...)

void picpack(PICFILE *pf, char *pix, char *fmt, ...)

void picerror(char *string)
```

DESCRIPTION

These functions read and write raster images in *picfile*(9.6) format. They are loaded by option `-lfb` of *2l*(1) et al. Open picture files are referred to by pointers of type `PICFILE*`.

Picopen_r opens the named picfile for reading and returns a pointer to the open file. If *name* is `"IN"`, standard input is used.

Picopen_w similarly creates the named image file for writing. The name `"OUT"` refers to standard output. *Type* is a `TYPE` attribute, as described in *picfile*(9.6); *x0* and *y0* are the upper left coordinates of the `WINDOW` attribute; *w* and *h* are the image width and height in pixels. *Chan* is a string specifying the order of channels for the `CHAN` attribute; the length of this string becomes the value of `NCHAN`. *Argv,* if nonzero, is conventionally the second argument of the main program; see *exec*(2). It becomes a `COMMAND` attribute recording the provenance of the file.

The special call `picopen_w(`*name*`, PIC_SAMEARGS(`*pf*`))` creates a file with the same attributes as an already open picfile. `PIC_SAMEARGS` mentions *argv* by name, hence the name must be in scope at the point of call.

Picread and *picwrite* read or write a single row of pixels using the character array *buf.* The length of the row is determined from the file's `WINDOW` and `NCHAN` attributes. One-bit-per-pixel images (of type `bitmap` or `ccitt-g4`, for example) are decoded to one byte per pixel, 0 for black, 255 for white, and are encoded as 1 for pixel values less than 128 and 0 otherwise. Files of type `ccir601` are decoded into conventional `rgb` channels.

Picclose closes a picfile and frees associated storage.

Wrpicfile copies a bitmap into a picture file. *Rdpicfile* allocates a `Bitmap` of given *ldepth* and reads picture file into it. Since `Bitmaps` are usually monochrome and only one or two bits deep, *rdpicfile* computes the NTSC luminance of the input image and uses Floyd-Steinberg error-diffusion dither to hide quantization errors.

Picputprop called after *picopen_w* but before *picwrite* adds header attributes, returning the revised `PICFILE` pointer.

Picgetprop returns a pointer to the value of the named attribute, or 0 if the picfile does not have the attribute. In both *Picputprop* and *picgetprop,* with multiple appearances (e.g. `COMMAND`) are expressed as a sequence of values separated by newlines.

The header file defines macros to extract commonly-used attributes:

```
PIC_NCHAN(pf), PIC_WIDTH(pf), PIC_HEIGHT(pf),
PIC_SAMEARGS(pf) (see picopen_w)
```

Picunpack extracts the channels of pixel array *pix* into separate array *args* of types described by the *fmt* character string. Format characters are `c`, `s`, `l`, `f`, `d`, for arrays of types unsigned char, short, long, float, and double. Format character _ designates a picfile channel to be skipped. *Picpack* reverses the process. These routines effect a standard machine-independent byte ordering.

Picerror prints messages for errors resulting from calls to *picfile* routines. (*Perror*(3) cannot describe some error conditions, like malformed header lines.)

EXAMPLES

Unpack the green and z channels from a file with channels `rgbz...`

```
PICFILE *pf = picopen_r("file");
extern char pixels[], green[][1000];
extern float zdepth[][1000];
for(i=0; picread(pf, pixels); i)
      picunpack(pf, pixels, "_c_f", green[i], zdepth[i]);
```

Reflect a picture about its vertical midline.

```
PICFILE *in = picopen_r("picture");
PICFILE *out = picopen_w("OUT", PIC_SAMEARGS(in));
int w = PIC_WIDTH(in);
int n = PIC_NCHAN(in);
char *buffer = malloc(w*n), *temp = malloc(n);
while (picread(in, buffer)) {
      char *left = buffer;
      char *right = buffer + n*(w - 1);
      for( ; left<right; left+=n, right-=n) {
            memmove(temp, left, n);
            memmove(left, right, n);
            memmove(right, temp, n);
      }
      picwrite(out, buffer);
}
```

SEE ALSO

picfile(9.6)

DIAGNOSTICS

Picread returns 1 on success, 0 on end of file or error.
Picopen_r and *picopen_w* return 0 for unopenable files.

BUGS

Picpack and *picunpack* store and retrieve floating point channels (types `f` and `d`) using native floating-point, rather than something transportable.

The code required to support `TYPE=ccir601` and the various `ccitt` fax compression types is missing!

NAME

3to1, mcut, improve, quantize, dither − picture color compression

SYNOPSIS

`fb/3to1` [`-e`] *colormap* [*input*]

`fb/mcut` [*input*]

`fb/improve` *colormap* [*input*]

`fb/quantize` [*input*]

`fb/dither` [*input*]

DESCRIPTION

3to1 approximates the full color (3 bytes per pixel) input picture file in one byte per pixel using the given *colormap.* If no *input* file is named, the picture is read from standard input. The `-e` option suppresses the default error-diffusion dither.

Mcut writes a color map, suitable for use by *3to1* on its standard output. The color map is computed using the median-cut algorithm and represents reasonably well, but not necessarily optimally, the colors of the input picture.

Improve reads a color map and a picture and writes on standard output a new color map that better represents the colors of the picture. Multiple passes of *improve* may produce better and better color maps.

Quantize is an *rc* script that packages all of the above to compress the full-color input image to one byte per pixel.

Dither likewise compresses full-color images to one byte per pixel. It uses a fixed color map that allows a speedy algorithm; *quantize* instead runs slower but gives better results.

SEE ALSO

picfile(9.6)

NAME

cmap, remap − map colors

SYNOPSIS

`fb/cmap` *colormap* [*input*]

`fb/remap` *colormap* [*input*]

DESCRIPTION

Cmap looks up the `rgb` channels of the input picture file (default standard input), in the *colormap*, writing the resulting image to standard output.

Remap is approximately the inverse of *cmap.* Pixel values in the input image are replaced by those that, when mapped through the input *colormap ,* come closest to reproducing the *input* image. The output picture includes a copy of the *colormap.*

SEE ALSO

picfile(9.6)

BUGS

Both commands work only if the input image contains `rgb` channels.

NAME

resample − resample a picture horizontally

SYNOPSIS

`fb/resample` *width* [*input*] [*B C*]

DESCRIPTION

Resample resamples the scan lines of its input image (default standard input) to the given new width. The image is decimated or interpolated using a well-designed cubic filter. See *transpose*(9.1) for assistance with vertical resampling.

The reference explains the optional filter parameters *B* and *C*. The default values give optimal alias rejection, and should not normally be tampered with.

SEE ALSO

picfile(9.6), ‘‘Reconstruction in Computer Graphics’’, by Mitchell and Netravali in SIGGRAPH 88 Proceedings.

NAME

rotate, transpose − re-orient an image

SYNOPSIS

`fb/rotate` *angle* [*input*]

`fb/transpose` [`−vhadrlui`] [`−o`*x y*] [*input*]

DESCRIPTION

Rotate rotates the image in its *input* picture file (default standard input) clockwise by *angle* degrees, writing the resulting picture file on standard output.

Transpose turns its *input* picture file on its side by reflection through its major (descending from left to right) diagonal, writing the resulting picture file on standard output. If no file name is given, the picture is read from standard input. Options yield all possible symmetries of the square grid:

`−d`	reflects the image through its descending diagonal (the default.)
`−a`	reflects the image through its ascending diagonal.
`−v`	reflects the image left-to-right through its vertical center line.
`−h`	inverts the image top-to-bottom through its horizontal center line.
`−r`	rotates the image to the right (clockwise) 90 degrees.
`−l`	rotates the image to the left (counterclockwise) 90 degrees.
`−u`	rotates the image upside down (180 degrees).
`−i`	identity transformation (for completeness only.)
`−o` *x y*	translates by (x,y). Without `−o`, the input and output files have the same upper-left corner.

Transpose is particularly useful to convince programs that work on the rows of a picture file to operate on columns. For example

```
fb/transpose big|fb/resample 48|fb/transpose|fb/resample 48 >tiny
```

makes a tiny 48×48 version of a big picture.

SEE ALSO

picfile(9.6), *resample*(9.1)

BUGS

Very large images may not fit in memory. The result of rotate is not anti-aliased.

NAME

dpic, twb − anti-aliased troff output to picture files

SYNOPSIS

`fb/dpic` [`-o` *list*] [`-w` *xmin ymin xmax ymax*] [`-d` *dpi*] [`-s` *stem*] [*file ...*]

`fb/twb` [*options*] [*files*]

DESCRIPTION

Dpic converts the output of *troff*(1) into anti-aliased images. Reading input from the named *files* (default standard input), it scan-converts characters using font outlines for the Merganthaler Linotron 202 phototypesetter; *troff*'s `-T202` flag will so apprise it.

Dpic puts each page of output in a picture file. Normally these files are named `page.1`, `page.2`, etc. Option `-s`*stem* causes page *n* to be put in file *stem* `.` *n.*

Option `-w`*xmin ymin xmax ymax* gives pixel coordinates of the rectangle in which type is to be set. The default is `-w 0 0 640 480`. Option `-d`*dpi* gives the mapping from *troff* coordinates (point sizes, etc.) to pixel coordinates; *dpi* is the number of dots (pixels) in the output files per inch of *troff* coordinates. The default is 100 dpi.

Option `-o` gives a list of pages to be processed. By default all pages of the input files are converted.

Dpic interprets several commands copied through using *troff*'s `\X'` ... `'` escape.

`\X'color` *r g b a*`'`
Set the foreground color to *r g b a.* The arguments are numbers between 0 and 255 specifying the red, green, blue and alpha (opacity) components of the color. Subsequent type and line-art will be set in this color.

`\X'bgcolor` *r g b a*`'`
Set the background color to *r g b a.* Before any type is set on any page, it is cleared to this color.

`\X'clear'`
Clear the page to the background color.

`\X'picfile` *name x y*`'`
Display a picture file with its upper-right corner at pixel *(x,y)*.

`\X'clrwin` *x0 y0 x1 y1*`'`
Set all pixels with *x0*<=*x*<=*x1* and *y0*<=*y*<=*y1* to the background color.

`\X'border` *x0 y0 x1 y1*`'`
Draw a one-pixel-wide rectangle in the foreground color. Opposite corners of the rectangle are at *(x0,y0)* and *(x1,y1)*.

Twb, the Toastmaster's Workbench, is an *rc* script that provides a rudimentary slide-making interface to *dpic* . It runs the *files,* (default standard input) through *grap, pic, tbl, eqn, troff* and *dpic,* passing any *options* to *troff* or *dpic,* as appropriate.

Twb loads the `-mtwb` macro-package into *troff.* The following macros mostly repackage the nonstandard *dpic* commands described above:

`.CO` *[r g b a]*
Set the foreground color. If the arguments are omitted the color reverts to its previous setting. Initially the foreground color is (255,255,255,255) (opaque white.)

`.BC` *[r g b a]*
Set the background color. If the arguments are omitted the color reverts to its previous setting. Initially the background color is (0,0,0,0) (transparent black.)

`.CL` Clear the page to the background color.

`.IN` *picfile x y*
Include a picture file, placing it with its upper-left corner at pixel *(x,y)*.

`.CW` *x0 y0 x1 y1*
Clear a window to the background color.

`.BW` *x0 y0 x1 y1*
Draw a window border using the foreground color.

`.SL` *[title]*
Start a new slide with the given title.

SEE ALSO

picfile(9.6)

NAME

xpand, picnegate − adjust dynamic range

SYNOPSIS

`fb/xpand` [`-s`] [*input*] [*lo hi* [*inlo inhi*]]

`fb/picnegate` [*input*]

DESCRIPTION

Xpand linearly adjusts the dynamic range of the *input* picture (default standard input) mapping value *inlo* to *lo,* and *inhi* to *hi. Lo* and *hi* default to 0 and 255. If *inlo* and *inhi* are not specified, the lowest and highest pixel values in the input image are used. By default, then, *xpand* expands the image's dynamic range by mapping its smallest pixel value to zero and its largest value to 255. Option `-s` causes all channels of the input image to be considered together when computing default values for *inlo* and *inhi,* thus preserving the hue of `rgb` pictures. Otherwise, each channel is treated separately. Option `-s` has no effect if *inlo* and *inhi* are specified on the command line.

There is no requirement that *lo* be smaller than *hi,* or that *inlo* be smaller than *inhi,* nor that any of those values be in the range 0 to 255. Output values not in the range 0 to 255 are clamped. For example,

```
xpand 0 255 255 0
```

inverts the pixel values of its input. For convenience, *picnegate* is a script that executes this command.

SEE ALSO

picfile(9.6)

NAME

intro − circuit design aids

DESCRIPTION

Circuit Design Aids (CDA) is a collection of programs used for the design and fabrication of electronic circuits. CDA is composed of programs that communicate through text files. A thorough introduction to CDA is given in the document "Circuit Design Aids (CDA) on Plan 9". What follows here is an abbreviated version.

Schematic entry

Schematics are created with the schematic editor *graw*(10.1). There are three varieties of schematic entities: library shapes (for simple gates), boxes of user defined parts and wires. The boxes and library shapes contain the symbolic *names* of pins which will be turned into pin numbers later on. The output of *graw* is an ASCII file in *graw*(10.6) format. This must be interpreted by *gnet*(10.1) to generate a net list. Net lists are combined into a common net list by *cdmglob*(10.1). *Cdmglob* also expands macros, interprets bus notation and matches the symbolic pin names in the schematics with the numeric pin numbers in a "pin file" that gives the correspondence between symbolic names and numbers in CDL format.

Programmable Devices

Besides commodity parts like the 7400 series, schematics may also use programmable devices such as PALs, Actel and Xilinx parts. CDA contains tools that convert logic equations written in a language called *lde*(10.6) format into the various formats required to fry the fuses on a programmable device. *Lde* format is interpreted by *lde*(10.1) and generates "symbolic product terms". Programs called "fitters" attempt to squeeze *lde* output into the selected programmable part. *Part*(10.1), *npart*(10.1) and *xpart*(10.1) are the fitters used for PAL-like devices. *Act*(10.1) is the fitter for Actel devices. The output is fed to *adil*(10.1) which in turn must be converted by the Actel software. The route to Xilinx parts is similar. *xil*(10.1) generates the intermediate format needed by the Xilinx software.

Physical output

The output of *cdmglob*(10.1) has no information about the actual physical construction of the circuit. A separate design subsystem of CDA known as *fizz* does the hard work of generating the information needed to build the circuit. *Cvt*(10.1) converts from the CDL output of *cdmglob* into *fizz*(10.6) format. *Place*(10.1) is a graphical tool that helps to position parts on the board. It requires a *fizz* description of the packages, net lists and the board. Finally, given the input to *fizz* and the output of *place* (a position file), *wrap*(10.1) will generate a wrap file containing the coordinates of all the wires.

File naming conventions

There is a strong convention for naming the files; it is highly encouraged but not enforced:

`.g`	schematic board description file produced by *graw*(10.1).
`.w`	netlist, output from *gnet*(10.1).
`.cdl`	circuit description language, output from *cdmglob*(10.1).
`.fx`	fizz netlist, output from *cvt*(10.1).
`.pos`	chip position file, output of *place*(10.1).
`.brd`	board description file including pinholes and special signal pins.
`.pkg`	package definitions.
`.pins`	pin definitions for input to *cdmglob*(10.1)
`.lde`	logic design equations, input to *lde*(10.1).
`.min`	*minterm*(10.6) output from *lde*(10.1), *quine*(10.1), *cover*(10.1), or *hazard*(10.1).
`.adi`	Actel design intermediate, output from `act -a`(10.1).
`.adl`	Actel Design Language, used by Actel software.

`.xy`	xymask, the venerable BTL film plotting language.
`.wx`	netlist, output from *cdmglob*(10.1).

SEE ALSO

A. G. Hume, M. Kahrs, and T. J. Killian, *Circuit Design Aids (CDA) on Plan 9*

NAME

act, adil − minterm to actel/simulator

SYNOPSIS

`cda/act` [*flag* ...] *file*
`cda/adil` *file*
simprog [`-d` [*id* ...]] [`-i` [*id value*] ...] [`-g`] [`-n` *iter*] [`-p` [*id period*] ...] [`-s`] [`-t` [*id value*] ...] [`-v` *var value*] ...]

DESCRIPTION

Act takes a *file* in *minterm*(10.6) format and produces a variety of output forms, selected by *flag* . If no *flag* is given, *act* constructs a balanced and factored tree of the circuit in the input *file* and covers it with gate patterns from the Actel FPGA gate library, printing the result in human readable form.

The basic Actel primitive is a two-level multiplexer, so *act* by default tries to express logic trees in terms of multiplexers prior to template matching. The `-m` flag reduces the zeal of this process.

The other *flags* select the output format. They are described in more detail below; here is a summary:

`-a`	print in Actel-like format suitable for *adil*.
`-c`	generate a C program for simulation.
`-d`	debug; just print the resulting tree.
`-f`*n*	set maximum fanout to *n* (default 10)
`-u`	unique; find common subexpressions
`-v`	verbose; include pin names in output

The `-u` flag causes *act* to output gate counts and usage (if default output is selected) or indicating fanout (for debug output). *Act* adds buffers when a signal or gate fanout exceeds the maximum specified by `-f`.

Adil takes output from `act -a` and produces Actel ADL format that can be fed to the proprietary Unix-resident Actel placing and routing software.

The C program generated by `act -c` can be compiled and loaded with loader options `-lsim -lg -lstdio`. The resulting executable is a simulator that displays signal traces in an *8½*(1) window and takes the following arguments:

`-d`	display the values of the named *ids.* Each *id* may be an identifier or an element of an *lde* array, in the form *id* `[` *n* `]` where *n* is an integer.
`-i`	set the initial value of the named *ids* to the corresponding *values.*
`-g`	suppress graphics; give textual output only
`-n`	set number of iterations for the simulation
`-p`	set half period of the named *ids* to the corresponding *periods,* forcing the *ids* to be clock signals.
`-s`	suppress screen graphics; output *pic*(1) input.
`-t`	specify the trigger values for the named *ids.*
`-v`	draw a guide edge when *id* attains *value.*

Given the number of details involved in running a simulation, simulators are typically invoked from *mk*(1). If the `-p` flag is not given, the simulator takes forcing input from standard input. The format for this input is identical to that of the output, so it is possible to run simulators in a pipeline.

Beware when catenating multiple *minterm* files to form input to *act*; the scope of external symbols in the input files is lost, so there can be name conflicts. On the other hand, this approach makes it easy to add an external environment to a simulation.

FILES

`/$objtype/lib/libsim.a` simulator driver

SEE ALSO

lde(10.1)

BUGS

Adil should be subsumed in *act*
Act doesn't check the fanout of buffers it adds.

NAME

annotate − annotate circuit schematic files with pin numbers

SYNOPSIS

`cda/annotate` *net-list net-files*

DESCRIPTION

Annotate takes the pin numbers from the net-list file (output from *cdmglob*(10.1)) and the net files (output from *gnet*(10.1) and stored in filenames ending in `.w`) and writes the coordinates of the pin numbers to corresponding files with a `.a` suffix. When combined with the corresponding `.g` files (or the older, deprecated `.j` files), circuit schematics will display pin numbers next to chips.

SEE ALSO

cdmglob(10.1), *graw*(10.1)

NAME

cdl − circuit description language

DESCRIPTION

The circuit descriptions used by the various circuit design aid programs are expressed in dialects of CDL — the circuit design language described below. This is half of a complete project specification; namely an electrical circuit with chips, pins and connecting signals. The other half, a physical layout with pins and chip positions, is described in *fizz*(10.6) format. The commands described below are recommended; others exist and may work but are regarded as obsolete.

Logic design

A circuit consists of *chips* connected by *signals.* The point of connection is denoted by a *pin.* Each chip has a *type* which describes its logical and electrical characteristics. (For example, `74S181` is a chip type.)

Types, signals, and chips are identified by name. Pins are identified by name and number. A *name* is a string of letters, digits, or any of the characters `+-.$/:<=>[]_`. Sometimes, the first character may not be a digit. A name may not be longer than 16 characters.

In the following description, literals appear as `constantwidth` and names are in *italic.* [ ] enclose an optional item and a list of items is written

{ *item* }

Commands are separated by either newline or semi-colon. A comment starts with a `%` and ends with a newline and may appear on any line. All white space serves only to separate tokens.

General

`.f` [*file*]
Subsequent input originated in *file* . If *file* is not present, the previous file name is restored.

`.q`
End of file.

Signal Description

signal [*pin-number*] [[,] *pin-name*]
name = *signal*
Lines that do not start with a period are signal definition lines. Signal definitions refer to the most recent `.c` command. The pin name and number refer to the chip.

Circuit Description

`.c` *name* [[`,`] *type*]
Instantiates the chip *name* with type *type.* This is typically used for I/O connectors. The command may occur more than once. The *type* of a chip need only be specified once in a circuit description. Signal descriptions that follow a `.c` command refer to pins on the chip.

`.c` *name* = *chip*
Establish *name* as a synonym for the previously defined *chip*.

`.m` *name1 name2*
Macro parameter definition. The signal *name1* is to be associated with macro parameter *name2.*

Chip Type Description

`.t` *name package* [*pin*] ...
Define a chip type *name.* The name of the *package* in which it is installed, and pin numbers, *pin,* for the special signal connections are specified. The special voltage pin numbers, if present, must be in the same sequence with which the special signals are numbered. This usage is discouraged; use the `.t[tT]` commands described below. (See `.v` command.) All commands of the form `.t`? are meant to follow a `.t` line.

`.t` *name* = *type*
name is a synonym for *type* .

`.tt` *sequence_of_single_character_pin_descriptors*
The number of characters must equal the numbers of pins on this *type*. The meaning of the descriptors is given in *smoke*(10.1).

`.tT` *sequence_of_single_character_pin_descriptors*
This means the same as the equivalent `.tt` command except that every `[gvwxyz]` pin must have a corresponding `.vb` pin.

`.tp` *name number ...*
The given pin *name* is associated with the pin *number. Name* may contain generators such as `Q[0-7]` which cause pin names `Q0 ... Q7` to be assigned to the pin numbers given. Multiple bracket constructs may be used. In any case, the resulting list is lexicographically sorted before assigning to pin numbers.

SEE ALSO

cdm (10.1)

NAME

cdmglob, cdmglob.errors − expand circuit macros

SYNOPSIS

`cda/cdmglob` [`-L`] [`-f`] [`-k`] [`-v`] [*files*]

`cdmglob.errors` *error_file_from_cdmglob*

DESCRIPTION

Cdmglob reads a circuit described in CDL from the specified files. Macro calls are expanded and pin numbers are substituted for pin names. The expanded CDL is printed on the standard output. The error output is used for diagnostics. Note that names in CDL are restricted in length so that care should be taken to keep macro names short. The options available are:

`-L`	generates LSL instead of CDL.
`-`	means standard input.

The `-f` option causes macro calls to be recursively expanded in-line. The `-k` option causes shape instances to be renamed to the name of the first actual output argument. The `-v` option causes the name of the pin to be output on the line. This is needed for *annotate*(10.1).

Macro Definitions

A macro definition corresponds to a file containing CDL. The name of the file for a definition *d* is *d*`.w`. Such CDL files may be produced using *gnet*(10.1). Macros may have signal names as parameters. These parameters are identified by a pin name. The (set of) formal signal names associated with the macro pin is replaced when the macro is called with actual signal names, unless the formal signal name is global. In this latter case the actual and formal signal names must be the same.

Macro calls

A chip of type *d* is a macro call if the file *d*`.w` exists. If no such file exists, the chip is assumed to be primitive (as in, say, `74S181`), and if the type is surrounded by <> brackets, the chip is an input output connector. If *d*`.w` exists then it is the definition of the macro *d*. Signal parameters of the macro are drawn in the same way as signals are connected to a chip. The pin name is the macro parameter name and the signal is the actual signal parameter. The name of the chip is interpreted as a macro name. A given macro can be called more than once, different instances being generated by different macro call names. Macros may not be called recursively.

Names

Signal, chip and pin names consist of letters, digits and the characters `+-/$`. Names of individual signals in a bundle or of chips in a group may also be generated: `name[ac-f]` generates `namea namec named namee namef`; `name{a,c,d,e,f}` will do the same thing but can be longer than one character. `name<i:j>` generates `name`*i* ... `name`*j* where *i* and *j* are represented in decimal as strings, all the same length. Thus, `BUS01` (and not `BUS1`) is in the set `BUS<0:15>`.

The set of generated names can be separated by an amount `k` by writing `name<i:k:k>` and multiple indexing is allowed: `name<i:j><p:q>`. Mixing the two generation methods is allowed.

Signal and chip names have scope local to a macro definition unless the name contains a `/`. A name containing a `/` is available throughout a circuit. Connector names are also available throughout a circuit. Signal and chip names used as formal parameters in a macro definition are replaced during macro expansion with the sequence of macro call names separated by `/` and ending with the actual parameter signal name.

Name Matching

The names of pins, signals and chips may also be generated from patterns. A pin pattern searches all pin names for the chip type. Signal and chip patterns search all signal or chip names. Patterns have the following form.

`*`	matches any sequence of characters
`[...]`	matches any of the characters enclosed
`[x-y]`	matches any character in the (ASCII) range `x` to `y`
`?`	matches a single character

Signal Expansion

A signal bundle may be connected to one or more chips (or macro) without having to write each chip or signal explicitly. In general each such array is expanded by generating the specified set of names. These names are then sorted alphabetically. The first signal is connected to the first pin of the first chip. Subsequent signals are connected to successive pins. If no more pins exists then the first pin on the next chip is used. The signal bundle must always end on the last pin of a chip and there must be no signals unattached at the end.

Cdmglob.errors takes the error output from *cdmglob*(10.1) and finds the real error by looking into the offending `.w` files and prints the error on standard output.

SEE ALSO

annotate(10.1), *cdl*(10.6)

NAME

drawp − draw board layout

SYNOPSIS

`cda/drawp` [*options...*] [*files...*]

DESCRIPTION

Drawp reads files produced by the *fizz*(10.1) suite of programs and writes output suitable for *pic*(1). Options are:

`−c`	Draw pins associated with chips.
`−d`	Draw datums.
`−H`	Omit *troff*(1) header.
`−k`	Display package names.
`−l`	Plot in landscape mode.
`−p`	Plot pins from the board definition.
`−r`	Do not attempt to rotate text when labeling chips.
`−t`	Display chip types.
`−v`	Show special-signal pins.
`−h` *file*	Plot holes as given by *file*, which is in the XY format accepted by ICON.
`−w` *file*	Plot nets as given by *file*, which is in the XY format accepted by ICON.

SEE ALSO

fizz(10.1), *pic*(1), *troff*(1)

NAME

findparts, ics, lookup, pins − find and manage parts

SYNOPSIS

`cda/findparts` [*file* ...]

`cda/ics` *part* ...

`cda/pins` *library part* ...

`cda/lookup` [`type`|`chip`|`index`] *part* ...

DESCRIPTION

Findparts reads the output of *getparts* (see *fizz*(10.1)) and finds the bin number of each part (or functional equivalent) in the local stockroom.

Ics searches the stock list to find the bins for the *part* arguments. The arguments should be the name of a chip, such as `74F00`.

Pins searches the standard pins file to find the pin names of the *part* and its relevant pin numbers. if three arguments are given instead of two, then the second argument is used as the pin library filename.

Lookup searches the chip database in one of three modes: `type` searches the database by type, e.g. driver or dsp. However, certain names are common (driver is one of them). Mode `index` should be used first to see all the fields for a given chip type. After filtering the output, then use the "type" option to find all chips of that type. Mode `chip` looks up all chips with that string, e.g. `2901` will find all `2901` chips.

FILES

`/n/coma/usr/ucds/lib/stock stock list /sys/lib/cda/lib.pins system pin library`

BUGS

The equivalence classes known to *findparts* are fairly crude.

The string matches are done exactly with *grep*(1); an inverted index would be better.

The CAPS database is much better than *lookup*, but is not on line.

NAME

artwork, check, clip, cvt, draw, drills, getparts, kollmorgen, list, pkgplot, place, prance, ring, signal, saf, wrap − physical layout programs

SYNOPSIS

`cda/artwork` [*option*] *file* ...

`cda/check` [`-uw`] [`-c`*chip*] *file* ...

`cda/clip` [`-f`*clipfile*] [*file* ...]

`cda/cvt` [*file* ...]

`cda/draw` [*option*] [*file* ...]

`cda/drills` `-d`*diams* *file* ...

`cda/getparts` *file* ...

`cda/kollmorgen` [`-hnbx`] *file* ...

`cda/list` *file* ...

`cda/pkgplot` [`-bp`] *file*

`cda/place` [*file* ...]

`cda/prance` [*file* ...]

`cda/ring` [`-lqsuvadk`] [`-z`*argument*] [`-w`*argument*] [`-c`*argument*] *file* ...

`cda/saf` [`-sdru`] *file* ...

`cda/signal` [*option*] [*file* ...]

`cda/wrap` [*option*] [*file* ...]

DESCRIPTION

The *fizz* suite of programs handle all the physical aspects of creating a wire-wrap, buried micro-via or microwire board. All the programs take *fizz_format*(10.6) input; *cdl*(10.6) can be converted with *cvt*.

All of the programs can take multiple files; most of the programs require that the files form a board description. Normally, this is arranged amongst four files (with recommended suffix): the board and special signal layout (`.brd`), the chip, chip type and net descriptions (from *cdmglob*(10.1)) (`.wx`), the package descriptions (`.pkg`), and the chip positions (`.pos`). In general, if the file arguments are missing, standard input is used.

Artwork prints various artwork information for the board definition in *files* . The options are

`-a`	prints XY mask clump includes for all placed chips with artwork fields in their package definitions.
`-r`	prints bounding rectangle information for the microwire router.
`-s`	generate silk screen information for chip layout.

Check checks the syntax and consistency of the given *files*. The `-u` option causes the names of any unplaced chips to be printed. Option `-w` checks readiness for wrapping. Specifically, it checks that no net is too large; no chip pin coincides with an inappropriate special signal pin, and no chip pin appears on more than one signal. Option `-c`*chip* prints out detailed information about the named chip.

Clip takes a board description (in *files*) and a clip description file (*clipfile*) and checks that all of and only the clips specified are present. Clips are simply pins on a wirewrap board. Almost always they are directly connected to a signal plane. Clips do not exist in the rest of the *fizz* suite; they are simply special signal pins. Standard input is used if there are no file arguments. The output reports missing clips in a format suitable as part of a board description. The clipfile consists of directives (one per line) of the forms

`[ssig|pin]` *numbers* `[chip|type]` *identifiers*
`tt [chip|type]` *identifiers*

Clips are put on either specific chips with the given names (`chip`) or chips of specified chip types (`type`). The clips are put on either the specified pins (`pin`) or pins belonging to the specified special signals (`ssig`). The identifier `ALL` refers to all chips or types. Lines starting with a `%` are ignored. The `tt` directive means pins whose entry in the `tt` field of the type (or the chip's type) is one of `GVWXYZ`. For example,

`ssig 0,1 type 74F374 74F245`
: Clips on power and ground for all chips of type `74F374` and `74F245`.

`pin 3-6,9 chip widget`
: Clips on pins 3,4,5,6,9 on chip `widget`.

Numbers are specified as a comma-separated list, possibly including *lo*−*hi* ranges. A missing *clipfile* argument is taken as

```
tt type ALL
```

Cvt converts CDL format input and outputs it in *fizz_format*(10.6) format. If no *files* are specified, standard input is read. Typically, *cvt* is used to process the output of *cdmglob*(10.1). The options are:

`-f`	Don't do families
`-c`	Don't emit comments
`-n`	Don't emit names

Draw generates a *plot*(6) description of the board layout of *files*. Standard input is used if there are no file arguments. The options are

`-p`	Show pins (as circles).
`-t`	Show chip types rather than chip names.
`-k`	Show package names rather than chip names.
`-v`	Show special signal pins as *(n+3)*-gons where *n* is the signal number.
`-P`	Draw package descriptions in *pic*(1) format. Each drawing shows the package name, the bounding rectangle, a cross at the origin, and numbered pin locations.
`-f`	Draws the pin frame.
`-r`	Removes the ruler.

Drills takes a board description and a set of drill diameters (*diams*) and produces a wraplist (like that produced by *wrap*) with an entry for every pin whose diameter is in that set.

Getparts reads its input files and generates a part list on standard output.

Kollmorgen generates the input files needed for Kollmorgen's router. Output is to the standard output. The options are

`-n`	Produce nets
`-b`	Produce border (keepouts are also generated). Wiring area shouldn't be too unusual.
`-h`	Produce holes. Holes may be wired or not depending on the declaration.
`-x`	Produce correspondence between net names and net numbers

List makes a fairly complete parts list giving type, package, and comment followed by each instance of it with position, rotation, and board side. The options are:

`-b`	list burnable parts, like PALs.
`-t`	Special Terry Wallis switch
`-s`	Short output

Pkgplot generates a plot of the package(s) in the input. the options are:

`-p`	Generate Postscript
`-b`	Generate bottom up instead of chip down view

Place supports interactive chip placement on a board. It requires a Plan 9 terminal running 8½. The user interface is mouse-driven. The main menu items are

`select`	a submenu allowing selecting chips or signals by name. Signals are displayed in the way they would be wired by *wrap* (no `-3` support).
`view`	a submenu supporting zooming, panning, grid overlay and resolution.
`insert`	insert unplaced chips.
`place`	a submenu supporting manual placement, machine placement and machine improvement of placement.
`read files`	reset the world and read the given (blank separated) filenames.
`write file`	write out the chip positions. The filename conventionally should have a `.pos` suffix.
`exit`	finito.

Chips can be selected by button 1 or by the button 3 submenu. Selected chips can be edited by the button 2 menu.

Prance generates the input files needed for Cadence's prance router on standard output.

Ring reads a board description and analyses the `Wire 's` therein; these contain the actual route of nets including all the inflection points. *Ring* walks each net, and starting from each driver calculates the length of the net (to the farthest pin). Next, it calculates the gate capacitance and distributed line capacitance. The rise time of the driver is used to calculate the maximum length of the line. Any offending long lines are reported to the user with the computed impedance of the line.

Saf outputs the packaging data suitable for giving to the automatic placement machine at Lisle.

Signal gives information about signals in the board description in *files*. Standard input is used if no file arguments are given. By default, all signals are shown as sequences of *chip*`.`*pin*`,` one signal per line. Note that the lines for the ground and power signals are likely be very long. The options are

`-w`	Wrap (route) signals before printing.
`-s`*name*	Show the signal *name* as both *chip* . *pin* and board coordinates (one point per line). Unplaced chips have negative coordinates.

Wrap generates a wraplist fo rthe board description in *files*. The options are

`-3`	don't do TSP
`-n`	connect to noconnects
`-o`	one post wraps are OK
`-v`	verbosity
`-c`	cents instead of mils
`-x`	don't do wire wraps
`-r`	set root string
`-b`	turn on buried vias
`-j`	produce a `.br` file suitable for the buried microvia router
`-t`	make file for cb router
`-h`	produce a `.hn` file suitable for the buried microvia router

SEE ALSO

cdl(10.6), *fizz_format*(10.6), *saf*(10.6)

NAME

fizz − physical layout input language

DESCRIPTION

Fizz is a suite of tools to build circuit boards from a circuit description. This section describes the input format for the various *fizz* commands. Most of the UCDS tools produce files in *cdl*(10.6) format; these need to be converted into *fizz* format by *cvt*.

Concepts

Types, signals and chips are identified by name. Pins are identified by name and number. A *name* is a string of letters, digits or any of the characters `+-.$/:<=>[]_`. Sometimes, the first character may not be a digit. A name may not be longer than 137 characters.

The physical design consists of a *board* containing *pin-holes*. The description details the positions of the pin-holes and the position and orientation of the chips. I/O connectors may be considered as chips with unmoveable packages.

The coordinate system for the board has *x* increasing to the right and *y* increasing upwards. The origin is at the lower left corner; no coordinate should ever be negative. The circuit board and components mounted on it are described as rectangles. They are positioned so that their sides are parallel to one or other of the axes. Measurements are integers measuring 0.001 inch. Coordinates are expressed as pairs of integers separated by `/` with the *x* coordinate appearing first. All rectangular regions are half open; the upper and right edges are outside the rectangle.

Syntax

The input is a sequence of items. An item consists of a item-type followed by a number of fields. Multiple fields are indicated by a trailing `{` on the keyword line and terminated by a line containing a single `}`. Fields are a keyword followed by the value for that field. Certain values are spread over multiple lines between `{}` as described above.

It is sometimes necessary to provide a list of coordinates. Invariably each coordinate is associated with a numbered object (say, a pin number). A one coordinate list consists of the index number followed by its coordinates as in

```
28 1700/2500
```

A series of equally spaced and consecutively numbered coordinates can be described by giving the first and last coordinates and separating the two with − as in

```
28 1700/2500 - 30 1900/2000
```

Coordinate 29 is 1800/2250. If the index numbers are equally spaced but not consecutive a step size can follow the − as in

```
12 2000/7000 -9 147 2000/1000
```

This describes coordinates numbered 12, 21, 30, and so on. If a letter follows the coordinate specifications, it specifies the drill to be used for the pinholes. The known drill types are

A	33	B	34	C	39	D	42	E	50	F	62	G	106
H	107	I	108	J	20	K	110	L	111	M	112	N	113
O	114	P	115	Q	116	R	117	S	118	T	119	U	100
V	20	W	122	X	123	Y	124	Z	125				

Items

In the following descriptions, each item has a sample input defining all possible fields. Some fields are optional; mandatory fields are marked by `**` which is *not* part of the actual input.

```
Board{
 name board_name
 align 1600/2000 9600/1700 1400/7100 9600/6600
 layer signalside 1
 plane 1 + VCC 2000 2000 8000 8000
```

```
 datums 100/100 135 100/8000 45 10000/100 45
}
```

The board name is set to *board_name* . The alignment points are used by `wrap -s` to align the board in Joe's semi-automatic wire wrapping machine. All four alignment points must be given. The *layer* field associates a layer number with a name to be used in XY artwork output. The layer numbers `0` and `1` are the two outside layers. The `plane` fields represent signal planes for circuit boards. The format is *layer sense signame minx miny maxx maxy* . *Sense* is a character meaning add (+) or subtract (−) the rectangle for the signal *signame* . The planes can be viewed with *place*(10.1). Note that multiple signals can be present in one layer. The *datums* field sets the positions and orientations of the three datums (alignment marks for artwork). The orientation is the angle formed by the two squares in the datum.

```
Package{
**   name DIP20
**   br -600 0 9600 3000
**   pins 1 20{
     1 0/0 - 10 9000/0 V
     11 9000/3000 - 20 0/3000 V
 }
 drills 1 2{
     1 500/1500 - 2 8500/1500 V
 }
 keepout 0 - VCC -1000 -4000 10000 3400
 plane 0 - VCC -1000 -4000 10000 3400
 plane 0 + VDD -500 -3500 9500 2900
 xymask clump {
     arbitrary XY mask stuff
 }
}
```

Each package definition may have an arbitrary origin. The bounding rectangle `br` is used for placement; the values are ll.x, ll.y, ur.x, ur.y. The `drills` field is for mounting bolts etc; it does not affect placement. Both the `pins` and `drills` fields take a minimum and maximum pin number. Placement of a package involves both its pins and rectangle. The rectangle must not intersect any other placed package, and there must be a pin-hole for each of the pins. The `keepout` field looks like a plane definition (the sense is always set to −). Multiwire wiring will not enter the specified plane. The `plane` fields are similar to those in `Board` but are instantiated for every chip using this package. The `xymask` field denotes the clump name (*clump*) for this package and some optional XY mask input (used by *artwork*). The XY mask input has leading tabs deleted, not white space, as blanks are significant to XY mask.

```
Chip{
**   name miscinv
**   type 74F240
}
```

This simply specifies the chip type.

```
Type{
**   name 74F240
**   pkg DIP20
**   family F
 tt ii3i3i3i3gi3i3i3i3iv
}
```

The `tt` field must have a letter for every pin of the package. Any pin whose letter is one of `gvwxyz` or `GVWXYZ` will be automatically attached to special signal 0,1,2,3,4,5 respectively. Other letters are ignored (they are used by other tools).

```
Net port 4{
 select 8
 miscinv 14
 syncff 13
 ackff 1
}
```

Signal nets have the net name and number of points on the item line. All other lines are simple *chipname* , *pinnumber* pairs. Net descriptions are normally produced by *cvt*.

```
Route{
**   name port
**   alg hand
 route{
     ackff 1
     miscinv 14
     select 8
     syncff 13
 }
 layout{
     100/2000 A
     3450/2000 Z
     3400/2000 A
     3400/1000
 }
}
```

This describes the routing for net *name* . The algorithm must be one of `tsp` (normal travelling salesman), `tspe` (travelling salesman specifying one end), `mst` (minimal spanning tree), `mst3` (minimal spanning tree of degree three), `default` (whatever is specified in the *wrap* command) and `hand` (the exact order is given). The routing is a list of *chipname*,*pinnumber* pairs. The layout is a list of pin positions. The layout uses the drill field to specify both the width of the trace and as a control. The layout is actually a sequence of lists of connected segments; a pin with a drill of `Z` is the last pin in a particular list. The trace width for a list is taken from the drill of the first pin for that list. The above example describes a T-shaped layout.

```
Positions{
 select 3200/2300 0 0
 miscinv 4900/1700 0 0
 syncff 2400/2700 0 0
}
```

Specify the position data for each chip. Each line has the form *chipname coord orientation flags* . The orientation is the number of right angles clockwise to rotate the package. The following bits in *flags*, which should be initialised to zero, have a defined meaning:

`4`	this chip is unplaced
`8`	the bounding rectangle is ignored in placement
`16`	the pinholes are ignored in placement.
`32`	the names are ignored in the silk screen output.

```
Pinholes{
 1400/6900 3200 300 10 V
 6650/6900 3200 300 10 V
 1600/1700 8100 1000 10/30 V
 1600/2700 8100 1000 10/30 V
}
```

Each pinhole specification has the form *coord lx ly spacing diam* which defines a rectangular array of pinholes with diameter of *diam*. The lower left corner of the rectangle is *coord*, and the width and height are *lx,ly* respectively. The pins are placed *spacing* apart. If *spacing* is of the form *sx*/*sy*, the spacings in the *x* and *y* directions are set independently.

```
Vsig 0{
 name GND
 pins 96{
    1 1800/2100 - 16 9300/2100 A
    17 1800/3100 - 32 9300/3100 A
    33 1800/4100 - 48 9300/4100 A
    49 1800/5100 - 64 9300/5100 A
    65 1800/6100 - 80 9300/6100 A
    81 1800/6700 - 96 9300/6700 A
 }
}
```

This defines the special signals. The special signal number follows `Vsig`. Pins are numbered from 1; the number of pins is given in the `pins` field line. A warning is given if any pins are not specified.

```
Wires {
 level COMP
 net iod15  {
    8100/8500
    8100/8550
    8000/8650
 }
```

Wires specify the inflection points of a signal net. Each instance of a net creates a new *wire*. The level can also be specified, although it is ignored by the *fizz* tools.

SEE ALSO

fizz(10.1)

NAME

gnet − graw to net

SYNOPSIS

`cda/gnet` [`-k`] [*file* ...]

DESCRIPTION

Gnet converts *graw* files to *cdl_format*(10.6) format. In order for part names to be associated with a symbol, the part name must be *inside* the bounding box.

`-k`	Extend the bounding box by two grid points. Handy in analog drawings.

FILES

`/lib/graw/gates.g`	the standard gate file
`/lib/graw/analog.g`	analog parts file

SEE ALSO

graw(10.1), *cdmglob*(10.1)

NAME

graw − gnot graphic editor

SYNOPSIS

`graw` [`-f` *fontfile*] [`-g`] [*file* ...]

DESCRIPTION

Graw is a multi-file graphic editing program specialized for schematic entry. *Graw* drawings consist of lines, boxes, text objects, and instantiations of previously defined drawings called *masters.* The *graw* user interface differs from that of most 5620/gnot programs in that button 1 controls *all* graphical entry.

The `-g` flag invokes an experimental mode in which `graw` attempts to render in grey scale. The `-f` flag allows the user to specify a font for displaying text objects.

By default, pressing button 1 will create a line with one end fixed and the other end attached to the cursor as long as button 1 is held down. Objects other than lines can be drawn by prefacing a drawing operation with a button 2 `onesies`→ selection. Button 1 is also used for *grabbing* objects. Grabbing takes precedence over drawing, and *graw* evaluates every button 1 hit to see if there is something to grab.

Grabbing rules vary by object. For example, a box can be grabbed by pointing to its interior. A grabbed box will cause all objects inside or touching it to be grabbed also. Grabbing a box's corner will also grab objects touching the two sides of that corner. An object inside a box may be grabbed without grabbing the box. *Graw's* grabbing rules are meant to be intuitively obvious. The author apologizes for cases in which this is not true.

Graw keeps a "text point" at the last location of a button 1 hit. Typing to *graw* creates a text object at the current text point. A text object orients itself based on its surroundings each time it is typed at or moved. Typing a carriage return causes graw to move the current text point down one or two ticks, depending on the surroundings.

Buttons 2 and 3 contain editing and file oriented menus, respectively. The button 2 menu entries are `onesies`→ `(box, dots, macro)`, `inst`→ (*master list*), `sweep, slash, cut, paste,` and `scroll.`

`onesies`→	selects a non-line object to be drawn with button 1. You get at most one non-line object per onesie.
`inst`→	selects a master to be instantiated and attached to the cursor until any button is pressed.
`sweep`	uses a rectangle input with button 1 (N.B.) to grab a set of objects and drag them until any button is pressed.
`slash`	differs from `sweep` only in that rectilinear lines are first cut by the input rectangle.
`cut`	undraws and moves the object(s) last drawn or moved to the cut/paste buffer.
`paste`	attaches a copy of the cut/paste buffer to the cursor until any button is pressed.
`snarf`	is a `cut` without the undraw.
`scroll`	attaches the entire drawing to the cursor until any button is pressed.

The button 3 menu entries are `edit, read, write, exit,` and `new,` followed by the list of filenames currently being edited.

`edit`	prompts for a file name and reads in the file for editing. Backspace and control-W may be used to edit the name; a null file name aborts the operation.
`read`	prompts for the name of a master file, reads it in, and plants a reference to it in the current file. The names of the masters in the file are added to those in the `inst`→ menu for the current file, overwriting older definitions if necessary.

`write`	prompts for a file name (starting with the current file name). The non-null result becomes the new file name and the file is written.
`exit`	terminates the program. It may be necessary to type a character and/or move the mouse after *graw* exits to really exit.
`new`	creates a new, unnamed drawing for editing.

Selecting a file name selects the current file.

Gnet(10.1) produces *cdl_format*(10.6) files from *graw_format* files.

FILES

`/lib/graw/gates.g`	the standard gate file
`/lib/graw/analog.g`	analog parts file
`/sys/font/1/7/PA`	default font file

SEE ALSO

gnet(10.1), *graw_format*(10.6), *cdmglob*(10.1), *grawp*(10.1), *annotate*(10.1)

BUGS

Doesn't handle parse errors well.
Crashes when it reads two `'e'` (end of master) lines in succession.

NAME

graw − graw file format

DESCRIPTION

Graw_format files are simple. There is one primitive per line, each primitive indicated by a single-character identifier. All strings are enclosed in double quotes. Definition need not precede use, though in practice `graw` produces *ref* primitives first, and master definitions are seldom found outside libraries.

The argument to a *ref* (or include) command is searched for in the current directory and then in `/lib/graw`.

Syntax:

body: prim | body prim
prim: line | box | string | dots | macro | inst | ref | master
line: `l` point point
box: `b` rect
string: `s` chars disp point
dots: `d` rect
macro: `z` rect
inst: `i` chars point
ref: `r` filename
master: mstart body mend
mstart: `m` chars
mend: `e`
rect: point point
point: INT INT
disp: INT
chars: " STRING "

Graw string displacements are specified by five bit codes defined below:

```
/* string placement displacements */
#define     HALFX 1
#define     FULLX 2
#define     HALFY 4
#define     FULLY 8
#define     INVIS 16
```

Invisible *string*s are typically defined for masters with connection points. Though the text is usually not displayed or printed, the remaining four bits should nonetheless specify a proper displacement for the sake of back-annotation.

FILES

`/lib/graw/gates.g` the standard gate file

SEE ALSO

graw(10.1)

NAME

grawp − draw schematics

SYNOPSIS

`cda/grawp` [`−t`] [*files...*]

DESCRIPTION

Grawp reads files produced by *graw*(10.1) and writes output suitable for *pic*(1). Under the `−t` option, the *troff*(1) header is omitted, so the result may be more easily included in a document.

SEE ALSO

graw(10.1), *pic*(1), *troff*(1)

NAME

ipf, pga132a − Actel to CDA translation

SYNOPSIS

`cda/ipf` *filename ...*
`cda/pga132a`

DESCRIPTION

Ipf reads at least an adil_file, as produced by `cda/act -a`, and a pin_file, returned by the Actel placing and routing software and produces two files: package and type definitions in CDL format in *name*`.pins,` and a new input file for the Actel software in *name*`.ipf,` where *name* is the prefix of the first file argument. If *name*`.ipf` were given to the Actel software, it would return a pin_file with the same contents as the argument to *ipf*.

Pga132a provides pin name to pin number translations for the Actel PGA132 package.

EXAMPLES

If you are working on a PGA132 device, then a typical use would be

```
cda/pg132a | cda/ipf /fd/0 xmit.adi xmit.pin
```

SEE ALSO

act(10.1), *lde*(10.1)

BUGS

Ipf misses pins whose definitions change between the `.io` and `.o` sections.
This should all be one program.

NAME

layout, route, gview − generate and view PCB trace layouts

SYNOPSIS

`cda/route` [`-tm`] [*file* ...]

`cda/layout` [`-a`*stem*] [`-gldqrs`] [*file* ...]

`cda/gview` `-a`*afile* [`-begrR`*file*] ... [*file* ...]

`cda/googoo` [`-gd`] [`-c`*minx miny maxx maxy*] [`-w`*x y*] *file* ...

DESCRIPTION

These programs deal with laying out and viewing circuit board traces.

Route takes a board description and generates a signal layout with the `Route` primitive (see *fizz_format*(10.6)). *Route* will only route within the bounds of a layer with a name of `analog_route`. Currently, these routes are the same as generated by *wrap*(10.1), either an MST (`-m`) or TSP (`-t`, and default) route. Signals may cross; it is up to the user to edit the layouts generated so that they don't. For the intended domain of analog circuits, you probably want to edit the layouts anyway.

Layout takes a board description and layouts and produces various files in either outline form (`-d`, and default) or Gerber format (`-g`). By default, the copper layer is produced. The `-s` option generates the silkscreen layer. In this case, bounding boxes are drawn around drillholes and chips, chips are labelled with their name and type (if the `comment` field is specified for a chip, the contents of that field is used instead of the type) and a rectangle is drawn around the board boundary. If the `-a` option is specified, then certain auxillary files, whose names all start with *stem*, are generated defining drills and apertures and other administrative details. For Gerber outputs, if the `-l` option is given and the `Board` definition includes an `xymask` field, the contents of that field should be two numbers followed by a string; the string will be printed centered on the given point. The `-r` option applies to outlines only; it causes all corners to be rounded.

Due to the nature of Gerber plots, *layout* requires a particular style of drill specifications. The drill names `A` through `Z` inclusive are reserved for aperture designations. (Actually, `Z` is a control and not an aperture, see *fizz_format*(10.6).) Any apertures used should be defined in the `drillsz` field in the `Board` definition. The type letter should be either `r` (round aperture) or `s` (square aperture); in both cases, the size is the diameter of the aperture. Furthermore, any drill used for a pin should specify, as the type field, the drill used make the pad for that pin.

Gview displays the given Gerber files. It requires an aperture definition file as produced by *layout*. For each file displayed, you can set the colour (black is `-b`, grey is `-g`) and a horizontal reflection (`-r` means reflected, `-R` means unreflected). Normally the plot is scaled to fit the layer; the `-e` option sets the scale to one (exact size).

Googoo is another Gerber viewer that assumes the same apertures as *xymash* and displays the Gerber output at a factor of 10 magnification, or more precisely, .001 inch to one pixel. You can pan the layer around the plot by using the mouse (depress a button to select a point on the plot and release it when that point is in its desired position on the screen). Typing any character causes *googoo* to exit. You can set a clipping rectangle and the initial position of the layer by the `-c` and `-w` options respectively. The `-d` option causes nothing to be displayed; it is used for mainly for debugging.

EXAMPLES

These drill specifications give .1 inch wide traces and 200 mil pads around .062 drill holes.

```
Board{
        ...
        drillsz {
                a 62 M
                A 100 r             % regular trace
                M 200 r             % pad
        }
```

```
}
```

View the silkscreen for a board overlaying the copper shown in grey.

```
gview -a ex1.gerber -rg ex1.cu.ger -Rb ex1.silk.ger
```

FILES

`/sys/cda/gerber/ex1.*`	examples of inputs and outputs of the above programs.

BUGS

Route does a half-hearted job at best. Eventually, it should do a real route that guarantees no crossing nets.

NAME

lca2pin, pga132x − Xilinx to CDA translation

SYNOPSIS

`cda/lca2pin` *filenames* ...
`cda/pga132x`

DESCRIPTION

Lca2pin reads at least a *lca_file* returned by the proprietary Unix-based Xilinx placing and routing software and produces a package and type definitions in CDL format in *name*`.pins,` where *name* is the prefix of the first file argument.

Pga132x provides *lca2pin* with pin name to pin number translations for the Xilinx PGA132 package.

EXAMPLES

If you are working on a PGA132 device, then a typical use would be

```
cda/pg132x | cda/lca2pin /fd/0 xmit.lca
```

SEE ALSO

xnf(10.1), *lde*(10.1)

NAME

LDE, lde − logic design equation programs

SYNOPSIS

`lde` [*option*] [*filenames*]
`LDE` [*option*] [*filenames*]

DESCRIPTION

Lde is the front end of a set of programs that prepare data for fuse-programmable logic elements. It accepts an expression language described in *lde_format*(10.6) on its standard input or from the named files, and writes an interpretation on its standard output. *Lde* produces minterms in *minterm*(10.6) output which may be reduced by *quine*(10.1) and *cover*(10.1). The options are:

`-o` Produce octal output (rather than the decimal default)

`-x` Produce hex output.

`-L` Put a the sum of products representation of the output on the standard error file.

`-d -v -T`
Produce other stuff to help debug the program.

Numeric parameters may be passed from the command line with

−*n* where *n* is decimal. The (zero based) `m`*th* numeric parameter is substituted for the symbol `$m` in the input.

LDE is an analog of *lde* that has some restrictions on the input specification, produces a cover that is often, but not necessarily minimal, but runs much faster. *LDE* takes some more options:

`-L` Output a sum of products representation to standard error.

`-I` Also calculate the complements of the logic functions.

`-X` Also calculate the xor of the output signals with their logic functions (useful for programmable parts that support toggle flip-flops).

SEE ALSO

lde(10.6), *minterm*(10.6), *cdl*(10.6), *quine*(10.1), *xpal*(10.1), *urom*(10.1), *cdm*(10.1)

BUGS

LDE does not support 'don't cares' or multiplication, division, modulo, or right and left shifts by variables.

NAME

lde − logic design expression language

DESCRIPTION

The *lde* language contains declaration areas that must appear in the following order:

`.x`	an optional chip declaration area,
`.m`	a master pin definition (LDE only)
`.i`	an input declaration area,
`.o`	an output declaration area,
`.io`	an external input/output declaration area
`.b`	a buried input/output declaration area
`.f`	an optional field declaration area,
`.e`	and an expression area.

The *lde* language contains expressions like those in C. Identifiers may include +, −, and `.` and semicolons are not used to end statements. Symbols must be declared before used.

The chip declaration area contains two strings, *name* and *type.*

Variables are declared by white-space delimited lists in the `.i`, `io`, `.b`, or `.o` areas or by appearance on the left of an = in the `.f` or `.e` areas. The variables are computer words with one or more bits representing two-level logic signals. In the default case, the least significant bit represents a single signal. An entry *identifier* `[` *n* `]`, where n is an integer, maps the logic signals *identifier0*, *identifier1*, ... *identifiern*−1 to the least significant through the *n*−1*th* bit of *identifier.* The numeric suffixes are left filled with zeros so that they all have the same number of digits. Similarly, an entry in the field declaration area of the form *n_id* = *o_id o_id* ... defines a new multibit variable *n_id* the least significant bit of which is the first old identifier, *o_id* and the higher bits the following old identifier.

Lde also accepts white-space delimited declarations of the form *name* **:** *master* in the `.m` area to declare an instance of a master definition. The master corresponds to a library definition in the target technology to be used. Variables of the form *name.id* for each pin in the corresponding master library definition may then be used in the expressions.

In the `.e` area, the binary operators `*`, `/`, `%`, +, −, <, <=, >, >=, ==, `!=`, `&`, `^`, `|`, `&&`, and `||` have the same meaning as in the C language. So do the unary operators `!` and ~ and the conditional operator `?:`. Since *lde* is an expression language, no flow control (such as `if` or `switch`) is allowed. An expression selector is available; *expra*`{[[`*exprb*`]:]`*exprc*`, [[`*exprd*`]:]`*expre*`,` ... `}` has the value of *exprc* if *expra* equals *exprb*. If there is no *exprb* and there is a colon, then *exprc* is the default case. If there is no *exprb* and no colon, then the pre-incremented value of the prior value of *exprb* is used, and the prior value of *exprb* is initialized to −1.

Combinatorial logic may be specified with the assignment operator, `=`. The assignment operator ~= is a cue to down stream programs that the combinational logic of the right hand side is to be inverted.

The assignment operators `:=`, `#=`, and ^= specify clocked outputs. The expression on the right hand side is the clock. The data input is a simple assignment statement as above. `:=` means D flip-flop, `#=` transparent latch, and ^= toggle flip-flop (output toggles when the date input is true). Optionally for clocked devices, +=, −=, and `%=` define signals that set, clear, and qualify the clock.

The operator `*=` assigns the enable expression for tri-state outputs. Sometimes, in the case of tri-state outputs used as inputs, it is important to state whether the input is before or after the tri-state driver. *id*<−*P* means use signal *id* at the pin (after the tri-state driver) while *id*<−*Q* means use the internal signal before the tri-state driver.

Identifiers may be modified by a appended single quote (`'`), in which case a value of one has the meaning "don't_care" for the unmodified identifier.

Numeric values may be passed from the command line of the program interpreting the *lde* language. They appear as `$m`. The (zero based) m*th* occurrence of −*n* on the command line substitutes the value *n* for the symbol `$m`.

EXAMPLES

A 4-bit counter.

```
.i
     ck
     en
.b
     x[4]                /* a buried vector */
.o
     c[4]
.f
     rx = x3 x2 x1 x0    /* note use of vector elements */
.e
     x = x ^ (x + 1)     /* expression for counter */
     x ^= ~0*ck          /* if the elements are toggles */
     c = rx              /* output bit reversal of counter */
     c *= ~0*en          /* note ~0* idiom */
```

A simple state machine that indicates even, state equals 3, or odd, state equals 0, number of input ones.

```
.i
     input clk reset
.io
     state[2]
.e
     state = state {
                 0:
                       (input == 1) ? 4 : 0,
                 4:
                       (input == 1) ? 0 : 4,
                 :
                       0
     }

     state' = state & 1            /* don't cares */

     state -= reset ~0 : 0         /* clear */

     state := clk ? ~0 : 0         /* d flip-flop */
```

SEE ALSO

lde(10.1)

NAME

minterm − minterm file format

DESCRIPTION

The *minterm* file format consists of at least one binary valued function definition. A function definition begins `.o` *name*[@*flag**] [*name*]... followed by line(s) that have the form *term*:*mask* ... *Name* is either a string or a number. The first *name* following `.o` is a symbol of the function (usually an output pin name or number of a ROM/PAL/FPGA integrated circuit). Any other name*'s* are symbols of input binary variables. *Term* and *mask* are decimal numbers.

There is a correspondence between the bits of the numbers in binary representation and the input symbols, the first input symbol is associated with the least significant bit. The meaning of a bit with value 1 in *mask* is 'do care', and the meaning of a bit with value 1 in *term* is 'input must be 1'. Thus the *term*:*mask* is an implicant, and a set of them when *or*'ed together describes the input conditions for which the output symbol will have a value of 1.

For example:

```
.o 3 1 2
3:3
.o 4 1 2
1:3 2:3 3:3
.o 5 2 3
1:3 2:3
.o 11
.o 9
0:0
```

Output 3 is the *and* function of inputs 1 and 2; output 4 is the *or* function of inputs 1 and 2 (*quine*(10.1) would change this to 1:1 2:2); output 5 is the *exclusive-or* function of inputs 2 and 3; output 11 is a constant 0 and output 9 is a constant 1.

Flag information is essentially communication between *lde* and the technology mapper for a particular architecture, and is passed through unchanged by *quine*, *cover*, and *hazard* (see *quine*(10.1)). Among the *flags* are

`b`	buried
`d`	D flip-flop clock
`e`	output enable
`g`	clock enable
`i`	inverted sense
`t`	T flip-flop clock

Another example:

```
.o x0@b
0:0
.o x1@b x0
1:1
.o x2@b x0 x1
3:3
.o x0@t ck
1:1
.o x1@t ck
1:1
.o x2@t ck
1:1
.o c2 x2
1:1
.o c2@e en
```

```
 1:1
```

`x[012]` are bits of a buried three bit counter, each bit is toggled by `ck` only if all lower bits are 1. `c2` outputs `x2,` the output enable is controlled by `ck.`

SEE ALSO

lde(10.1), *quine*(10.1), *cover*(10.1), *hazard*(10.1), *xpal*(10.1), *part*(10.1), *act*(10.1)

NAME

mkpins, mkpkgs, getpkgs, mkpos /- select pin definitions

SYNOPSIS

`cda/mkpins` *file* `.wx` [*file* `.pins` ...]

`cda/mkpkgs` *file* `.wx` [*file* `.pkgs` ...]

`cda/getpkgs` *file* `.wx` [*file* `.fizz` ...]

`cda/mkpos` *file* `.fx`

DESCRIPTION

Mkpins sorts through the various pins files given and only selects the needed pins definitions by looking up the type from the `.t` line in the various `.pins` files. It writes the needed pins on the standard output and undefined pins on the standard error.

Mkpkgs is like *mkpins* except it accepts a list of package files and generates the appropriate package files on the standard output.

Getpkgs is like *mkpkgs* except it takes a CDL input (`.wx`) file and a *fizz* package file.

Mkpos creates a null position file from a *fizz* input file.

BUGS

CDL should go away.

NAME

npart − configure multi-bank programmable logic devices

SYNOPSIS

`npart` [*options*] [*filename*]

DESCRIPTION

Npart accepts logic functions described by *minterm*(10.6) as input and attempts to implement them with multi-bank programmable logic devices such as the AMD MACH series parts. It assigns pins and routes the interconnection switch matrix. Npart produces two output files *filename*`.m` and *filename*`.p`. The first is used by *xpal (10.1)* for programming the device; the second contains pin assignment information.

The options are:

`-t` *device_type*
use *device_type* as the target device programmable logic device (presently AMD MACH M110, M120, M130, M210, M220, and M230). By default, *npart* uses the type on the `.x` line of the *lde_format*(10.6) source file if present.

`-N` *n* Iterate at most *n* times (default is 5).

`-M` *n* Iterate an additional *n* times after successfully fitting the equations to the target devices to further reduce the interconnection wiring.

`-p` Use the pin assignments in the pin assignment files.

`-L n` Backtrack n levels deep in attempting to route the interconnection switch matrix (default is 2).

`-S n` Leave at least n unused inputs on each of the internal banks.

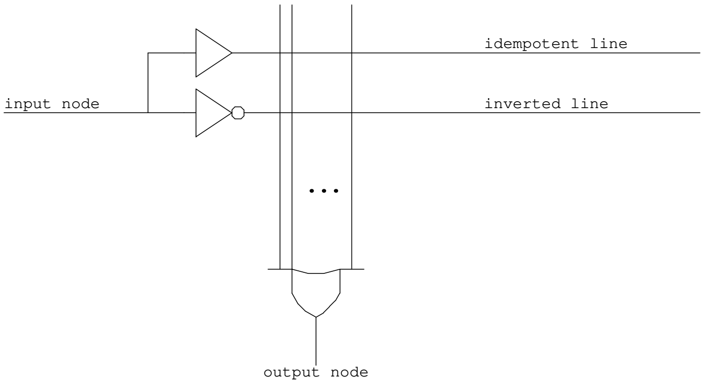
NAME

paddle − pal description language

DESCRIPTION

Paddle is a description language for detailing the fuse format of programmable devices. *Paddle* is used by *xpal*(10.1) to create the fuse map that *urom*(10.1) and friends want. It is also used by *xpart*(10.1).

Paddle has an underlying model of arrays of fuses. This model looks something like this: (`.,.`)... 0.000i 1.875i 3.625i 0.000i



The input to an *and/or* array is a set of nodes; the output is also a set of nodes. A node can be externally visible (such as a pin) or it may be an internal node. The buffered or inverted version of a node is called a "line". The intersection of a line with the input term of the *OR* is controlled by a *fuse*. The collection of fuses over a set of input and output nodes is called an *array*.

It is possible that a device may have multiple *and/or* arrays; examples of such devices include the Advanced Micro Devices Mach parts, the Cypress 7C361 finite state machine controller and the Signetics Macrologic (PLS 501, 601 and 701) parts. Therefore, every array declaration must be followed by a symbolic name. This helps `xpal` and `xpart` to give useful error messages. Also, each array must have an "offset" declaration. This permits the array to be placed anywhere in the fuse space. Next, the declaration of input and output nodes must be given. Note that in most programmable logic devices, input lines come in both complement and idempotent flavors and are provided courtesy of the buffer on the input pin. *Paddle* therefore has a declaration that declares that a given set of nodes are doubled, either the complement or idempotent line first.

Fuse arrays are used for declaring special fuses that some programmable parts offer. For example, the `22V10` has fuses that set the polarity of the output pin, whether the output is latched or combinatorial and so forth. These fuses are not organized in an *and/or* array and so *paddle* just permits a one-to-one mapping between artificial pin (external node) numbers and fuse numbers. permits multiple fuse arrays provided they are given unique names. Each definition begins by defining the name of the part along with possible synonyms. This is followed by (1) an array declaration (2) a fuse block definition (3) a type declaration (the `.tt` line) and lastly, a (4) pins declaration. The array declaration permits declaration of input and output pins to the array. The use of the *complement* keyword create 2 input lines for a given pin. The general form of a pin declaration is `pin:terms=fuse`, where *terms* is the maximum number of terms for the pin and *fuse* is the optional fuse number.

EXAMPLES

Here is part of the declaration of a 20L10:

```
20L10=NS20L10=AM20L10 {
    package "DIP24"
    declare {
      internal {
            output enables { 114..123 }
```

```
          }
          external {
                inputs { 1..13 }
                inverted outputs { 14..23 }
                ground { 12 }
                supply { 24 }
          }
        }
        array and/or {
          inputs complement+ {
                2,    1,
                .
                .
                .
                11,   13
          }
          outputs {
                123:1,
                23:3,
                .
                .
                .
                }
          }
    }
```

SEE ALSO

xpal(10.1)

FILES

`/sys/lib/cda/library.paddle`

NAME

part − configure common programmable logic devices

SYNOPSIS

`part` [*options*] [*filename*]

DESCRIPTION

Part accepts logic functions described by *minterm*(10.6) as input and attempts to implement them with programmable logic devices. It assigns pins partitioning them over multiple devices if necessary. In the case that a single device is sufficient, part produces two output files *filename*`.m` and *filename*`.p`. The first is used by *xpal (10.1)* for programming the device; the second contains pin assignment information. In the case of multiple devices *filename*`.`*nn*`m` and *filename*`.`*nn*`p` files are generated, where *nn* are integers between 0 and the number of devices minus one. In the case of multiple devices, a *filename*`.j` giving a schematic representation of how the devices are connected in *graw*(10.1) format is also produced.

The options are:

`-t` *device_type*
: use *device_type* as the target device programmable logic device. By default, *part* uses the type on the `.x` line of the *lde_format*(10.6) source file if present.

`-N` *n* Iterate at most *n* times (default is 5).

`-M` *n* Iterate an additional *n* times after successfully fitting the equations to the target devices.

`-p` Use the pin assignments in the pin assignment files.

NAME

pga − generate CDL pins format for large packages

SYNOPSIS

`cda/pga` *pin_file*

DESCRIPTION

Pga takes a list of pin names (in numerical order), one per line, and generates a pin list suitable for *cdmglob*(10.1) and friends. Each pin name may be optionally followed by a tab and a single character *smoke*(10.1) pin type. Special pin names are: `VCC`, `VDD`, `VSS` and `GND`. These are recognized and generate the proper pin type. Unless specified, the default pin type is `4`, or bidirectional tri-state.

There are two other special names: `NC`, which denotes no connect pins and `--`, which specifies a hole in the PGA. Comments may be preceded by a '# '. Special commands (preceded by a '! ') worth knowing about are: *debug*, *map*, *clip*, and *holes*.

`debug` is useful for double checking your definition,

`map mapfile` maps the pins via a map file; a map file is a 1:1 mapping suitable for PGA adaptors. An example of a map file that will reverse all the pins in a mysterious 5 pin package is:

```
1	5
2	4
3	3
4	2
5	1
```

`clip [v|g]` tells *pga* to capitalize the appropriate plane (suitable for *clip*).

`holes` counts holes (pins declared as `--`) as pins and therefore will generate more pins. Normally holes are ignored.

EXAMPLES

Here's an example:

```
# section E with hole
WE-	i
DI05
DI10
NC
--
--
--
VSS
--
--
--
DI38
DI34
REGADR4		i
CI6	i
```

SEE ALSO

cdmglob(10.1), *smoke*(10.1), *fizz*(10.6)

NAME

pll − phase lock loop calculator

SYNOPSIS

`cda/pll` [`−cfknoptzAPT`]

DESCRIPTION

Pll calculates the passive component values for a second-order phase lock loop. Component values and other calculations are emitted on the standard error output while the Bode plot is generated on the standard output.

The parameters of the loop are given on the command line:

`−c`	the capacitor, in farads
`−f`	the frequency of the loop in hertz
`−v`	the gain constant of the vco (in Hz/v)
`−n`	the divider multiplier
`−o`	the frequency of the loop ω
`−p`	the gain constant of the phase detector (in V/radian)
`−t`	the ‘type’ of the chip; should be in the list printed by the `−T` option.
`−z`	the damping constant ζ

Other output flags are:

`−A`	for an active filter (defaults to passive)
`−P`	for a Bode plot; output *grap*(1) format on standard output.
`−T`	will tell about the chip types known to *pll.*

This program is shamelessly stolen from Rhode’s book "Theory and Design of Digital PLL Frequency Synthesizers".

SEE ALSO

grap(1)

BUGS

Capacitor value should be in microfarads.
Failure to give all the values results in an invalid floating point fault.

NAME

quine, cover, hazard − logic programs

SYNOPSIS

`cda/quine`
`cda/cover` [`-s`]
`cda/hazard` [−*n*] ...

DESCRIPTION

Quine, *cover*, and *hazard* read the standard input and write the standard output, both in the format of *minterm*(10.6).

Quine produces a Quine-McCluskey reduction of the input data.

Cover does the covering problem, which is exponentially hard and may not finish in reasonable time. The `-s` option causes *cover* to not do the complete problem and go faster.

Hazard adds terms to eliminate internal hazards that can occur in PAL's and PLA's. The hazard can occur when the form of the equations is $(a\&x) \mid (b\& !x)$. When *a* and *b* are both true a glitch may appear on the output when *x* is changed. *Hazard* eliminates it by adding another term $(a\&b)$. If there are any −*n* option flags for *hazard*, only those outputs whose symbols are in the set of *n 's* will be modified by *hazard*, otherwise all outputs are (possibly) modified.

SEE ALSO

lde(10.1), *minterm*(10.6), *xpal*(10.1)

FILES

`qtmp`*n* and `bsort`*n* in the working directory for temporaries.

NAME

rework − diff two wraplists

SYNOPSIS

`cda/rework` [`-e`] [`-q`] [`-v`] [`-s`] [`-o`] [`-d`*net*] *old new*

DESCRIPTION

Rework takes two wraplists (the output of *wrap*(10.1)) and produces three wraplists: `UN.wr`, `RE.wr`, and `NEW.wr`. `NEW.wr` describes the result of removing the wires in `UN.wr` from *old* and then adding the wires in `RE.wr`. The list `NEW.wr` is electrically equivalent to *new* . Typically, the file *new* is generated by *wrap*(10.1) and *old* is the `NEW.wr` produced in the last rework.

The various options are

`-d`*net*	produce detailed debugging output. The optional netname *net* confines debugging to just that net.
`-e`	like `-v` except that input nets need not be connected.
`-o`	the nets in `NEW.wr` will be ordered. Normally unchanged nets are just copied.
`-q`	try to minimize the number of wires for the rework. Currently, this is only useful when the new net is strictly larger than the old net.
`-s`	print some statistics of the inputs.
`-v`	print a terse summary of the differences on standard output. `UN.wr`, `RE.wr`, and `NEW.wr` will be unchanged.

Rework ignores the start (`04`) and stop (`08`) bits in its input; it assumes all the wires for one net are sequential in the input.

SEE ALSO

wrap(10.1)

NAME

smoke − static circuit checks

SYNOPSIS

`cda/smoke` [`-l load`] [`-abcmnpsxL`] *files*

DESCRIPTION

Smoke reads and checks a circuit and reports simple errors like typechecking in C.

The circuit description language is *cdl*(10.6). The *files* are usually a circuit description file(s) made with *gnet*(10.1) and/or *cdmglob*(10.1) and the pins files with `.tt` lines like those given to *cdmglob*(10.1). Files with net lists come first, pins files come second.

The `.tt` line contains one character per pin on the chip according to the following table:

`1`	open collector output
`2`	totem pole ttl output
`3`	3-state output
`i`	input
`p`	pull-up (for 1)
`0`	1 and p
`4`	3 and i
`5`	1 and i
`6`	1 and p and i
`j`	p and i
`k`	d and i
`9`	voltage source
`v`	vcc sink (.vb 1)
`w`	.vb 2 sink
`x`	.vb 3 sink
`y`	.vb 4 sink
`z`	.vb 5 sink
`g`	ground
`n`	no connect (use as tie point prohibited)
`8`	analog output
`a`	analog input
`A`	analog input/output pin
`s`	switch contact
`t`	terminator
`b`	PAL undeclared pin
`I`	current source (not supported)
`J`	current sink (not supported)
`D`	+ driver
`d`	- driver
`R`	+ receiver
`r`	- receiver
`P`	pulldown
`.`	no type

The various options for *smoke* are

`-a` Don’t print out errors on analog nets.

`-b` Don’t print out errors on nets with bidirectional pins.

`-c` Print out the entire circuit with type declarations by each pin (but does *not* do any checking.

`−l`	takes an optional loading count; only nets with more loads will be flagged for load complaints.
`−m`	Don’t complain about lone pins on macro signals.
`−n`	Turn on ‘complete’ nets; when errors occur, nets will be completely printed out (when used with `−x`).
`−p`	Turn on paranoid mode; *smoke* normally doesn’t complain if there are any undeclared pins in a net. Now it will.
`−s`	Ignore multiple source messages. Not recommended for general use.
`−x`	Turn on extra (excessive) mode. Offending nets will be printed out in gory detail; *very* useful for debugging the nets it complains about.
`−L`	Ignore lonely pins with names beginning with `$` (local names).

BUGS

In *smoke,* the last definition of the chip is the one that counts.

NAME

stock − stock list

DESCRIPTION

The stock file is a plain text file. The first column is the part name, the second column is the bin (bins have the form <bin number><section><drawer>), the third column is the quantity and the remaining string is the chip description. The latest entries include the manufacturer at the end of the line in the form "[manufacturer]".

SEE ALSO

findparts(10.1), *ics*(10.1)

FILES

`/n/coma/usr/ucds/lib/stock`

BUGS

The quantity is seldom up to date.

NAME

swrap − generate control information for semi-automatic wiring machine

SYNOPSIS

`swrap` [*options*] [*file*] ...

DESCRIPTION

Swrap controls a semi-automatic wiring machine as directed by a `.wr` file generated by *wrap*(10.1).

Options are:

`-f`*name*
Use the file *name* instead of `/dev/eia0` to control the machine.

`-d`*mn* The two-character string *mn* sets the preferred direction for wire routing. *M* gives the first preference and *n* the second, according to the following code:

`0`	route from left to right (increasing X).
`1`	route from bottom to top (increasing Y).
`2`	route from right to left (decreasing X).
`3`	route from top to bottom (decreasing Y).

`-l` Produce a listing as a reference for the machine operator; it describes the wires in the sequence in which they are to be installed.

`-r`*d* The digit *d* specifies how the board must be rotated from the position implied by the Circuit Design Language definition of the board. The rotation is the number of right-angles by which it is to be rotated anti-clockwise, plus four if the board is first to be flipped over (X and Y coordinates interchanged). The initial rotation is given in the board definition.

`-v` Set verbose mode.

Unless a listing is requested, the on-line Standard Logic wire-wrap machine must be connected to the designated RS-232 port. The operator will first be required to calibrate the machine by moving the pointer to specified pin positions. Then the machine will point at successive pins which must be wired. The typed commands to which the program responds are as follows.

`udlr`	Move the pointer a small distance up, down, left or right. If preceded by a number scale the distance moved accordingly.
`s` *n*	Skip to wire number *n*.
`c`	Check the calibration by moving the pointer to the reference pin.
`C`	Check positions of all four corner pins of the board.
`v`	Change to and from verbose mode.
`q`	Quit after moving the pointer back to the reference pin.
`?`	Print details about the wire currently being installed.

FILES

`/dev/eia0`	RS-232 port

NAME

urom − read and write programmable devices through DATA I/O Unisite

SYNOPSIS

`urom` [*option*] ...

DESCRIPTION

Urom serves as an interface to the DATA I/O Unisite™ programmer. Options are used to specify the device type, and whether the device is to be read or written.

`-w` Specifies that the device is to be written; default is read.

`-s`*n* Specifies a starting address (default 0) in the device in decimal.

`-t` *string*
string Specifies the device type of the device. If it is an ambiguous name, all the possible conflicts are listed.

`-m`*string*
string Specifies the manufacturer of the device; this must be specified *before* the type (-t) of the device. Only needed if the name is ambiguous.

`-n` Causes the program to echo the code that it is sending to the DATA I/O, and causes the DATA I/O the echo the size and initial state specified by the code. No reading or writing is done.

`-b` Causes a blank check to be run, the illegal bit test is run unless it is an electrically alterable rom. No blank check is done if the rom is being read.

`-X`
`-x` Specifies hexadecimal data with upper case or lower case respectively.

`-C` Specifies character data, for logic devices with JEDEC format.

`-D`
`-d` Specifies decimal data, octal is default.

`-O`
`-o` Specifies octal data, which is the default.

`-fn` Use format *n 50*, the default, is for hex, octal, or decimal format files. 91 is for JEDEC files.

`-I` *string*
inhibits checking of following character string, e.g. '-I C' inhibits continuity checking.

`-J` JEDEC format (same as -f91 -C option)

`-i` Causes the data to be (ones) complemented on input and output.

`-v` Normally *urom* does it work silently, this is the verbose flag.

To read an Intel 2716 device one could say

```
urom -m Intel -t 2716 < filename
```

or to write a National PAL16L8 one could say

```
urom -w -m National -t 16L8/A/A2 < filename
```

The input for logic devices on the Unisite must be in JEDEC format. In this case use

```
urom -w -m National -t 16L8/A/A2 -f 91 -C < filename
```

or

```
urom -w -m National -t 16L8/A/A2 -J < filename
```

When a device is read, the addressed locations are copied, one per line, onto the standard output. When a device is written, the standard input is assumed to be of the same form, and is copied onto the device. Various Unisite errors, such as the device having a pattern that conflicts with the data (illegal bit test) when being written, are reflected back to the user. Transmissions over the RS-232 line are checksummed, and when writing the device is verified.

The speed of the Unisite should be set to 9600, position 14.

FILES

`/sys/lib/cda/urom.codes`

SEE ALSO

xpal(10.1)

BUGS

Case shouldn't be important for type or manufacturers.

NAME

xil, xnfpins, xnffrom, xnfto − xilinx tools

SYNOPSIS

`cda/xil` [`-x`] *name* `.m` > *new* `.xnf`
`cda/xnfpins` *name part* [*pin...*] > *new* `.pins`
`cda/xnffrom` *part old* `.pins` *new* `.xnf` > *new* `.pins`
`cda/xnfto` *part name* `.pins` *name* `.xnf` > *name* `.cst`

DESCRIPTION

Xil factors and translates input in *minterm*(10.6) format to Xilinx Netlist Format suitable for processing by the proprietary Xilinx program *ppr* (partition, place & route) and subsequent programs.

Xilinx hard macros and RAM/ROM symbols generated by the Xilinx *memgen* program can be used via the *lde*(10.1) `.m` facility.

Xnfpins, *xnfto*, and *xnffrom* create and maintain CDA pins and Xilinx constraint files. *Xnfpins* produces an initial `.pins` file given the Xilinx part number (e.g. `4005pg156`) and a list of statically assigned pin names, typically those used for initializing the part. Subsequent programs retain this initial information in the face of changes in automatically assigned pins.

Xnffrom takes `EXT` lines in an `.xnf` file produced by *lca2xnf*(10.1) as back annotation to update the corresponding CDA `.pins` file and subsequently constrain *ppr*'s choice of pins.

Xnfto takes `.tp` lines following `#float` in the `.pins` file that appear in the `.xnf` file and fixes them in the `.cst` (constraints) file used by *ppr. Xnfto* should be used only to maintain pinouts generated by *ppr* and *xnffrom* and *only* after said pinouts have been set in physical design concrete.

The files used and generated by these programs have to be shipped back and forth between Plan 9 and a suitably licensed Xilinx platform. Use *mk*(1) to control this.

FILES

`/sys/lib/cda/40`*nn*`.pin`

SEE ALSO

lde(10.1)
ppr(Xilinx)
memgen(Xilinx)
lca2xnf(Xilinx)

BUGS

It may be complicated, but Actel is worse.
Ppr gets very confused if it sees a constraint against using a pin it wasn't going to use anyway.

NAME

xnf − minterm to xilinx XNF translation

SYNOPSIS

`cda/xnf` *file*.m

DESCRIPTION

Xnf translates input in *minterm*(10.6) format to Xilinx Netlist Format suitable for processing by the Xilinx minimization, placing routing software. This translation is very naive and not guaranteed to be acceptable by the Xilinx *ppr* program for X4000 parts. In any case, the Xilinx *xnfopt* program should be run first.

SEE ALSO

lde(10.1)

BUGS

Use *xil* and the 4000 series parts instead.

NAME

xpal − data preparation program for PAL's and PLA's

SYNOPSIS

`cda/xpal` [− *option*] [*type* [*filename*]]

DESCRIPTION

Xpal reads data from the standard input or *filename* in the form of *minterm*(10.6) and writes on the standard output in a form suitable for any of the pal/prom burners.

The options are:

`−m manufacturer`
Specifies a manufacturer that will be output as a comment

`−t type`
Specifies a PAL type if not given by a .x line

`−l library`
Tells xpal where to look for library definitions. Defaults to /usr/ucds/lib/library.paddle.

`−v` Produce slightly verbose output (not harmful)

`−d` Produce debugging output

`−i` echoes the input

`−z` produces a zero checksum for SPRINT programmers

`−p` demands parsing; used in conjunction with -t, this can be used to debug new pal definitions.

The output is JEDEC suitable for any number of JEDEC compatible programmers. Xpal's principal advantage over pal is that the tables are now read in. The *type* flag on the command line or as given by the .x line in the lde file is used to address the correct tables.

The numeric symbol of a term that corresponds to the output enable of a pin is 100 + pin_number by convention. Just to be strange, the convention for the 22V10 is: 200 + pin_number for polarity, 300 + pin_number for architecture fuses and pins 25 and 26 are asynchronous reset and synchronous preset respectively. Each programmable device has its own mapping as defined by the pal definition.

The pals with X in their name use an additional convention. Since *quine* can only handle and-or logic, and the X pals have two different sets of and-or logic driving the two inputs of an xor gate, those two terms are number 20 + output pin number and 70 + ditto.

SEE ALSO

minterm(10.6), *paddle*(10.6)

FILES

`/sys/lib/ucds/library.paddle`

BUGS

The pin naming convention is peculiar at best.

NAME

xpart − partitioner (fitter) for PALs, PLAs and MACH parts

SYNOPSIS

`cda/xpart` [`−`*option*] [*type* [*filename*]]

DESCRIPTION

Xpart reads data from a stem *filename* in the form of *minterms*(10.6) directly from *lde*(10.1) and partitions the equations amongst a number of devices of the same flavor.

The options are:

`−N iterations`
Specifies the maximum number of iterations before failing

`−p` Use the existing pin files (.p) as the pin assignments

`−l library`
Tells where to look for library definitions. Defaults to /sys/lib/cda/library.paddle.

`−v` Produce slightly verbose output (not harmful)

`−g −D` Produce debugging output

`−i` echos the input

`−P package`
specifies the package

SEE ALSO

xpal(10.1) *minterm*(10.6), *paddle*(10.6)

FILES

`/sys/lib/cda/library.paddle`

BUGS

The pin naming convention is peculiar at best.