LynxOS Reference Manual

LynxOS Version 2.5
DOC-0168

Utility Programs (A–F)

Printed in the United States of America.

Contents

Utility Programs (A-F)

Utility Programs (A-F)

NAME

`Intro` – LynxOS utility programs

DESCRIPTION

The Utility Programs Reference Manual spans three volumes in its printed form: “Utility Programs (A–F)”, “Utility Programs (G–O)”, and “Utility Programs (P–Z)”. A master index for these three volumes can be found at the back of the third manual, “Utility Programs (P–Z)”.

The utility programs documented herein are all included as part of the LynxOS software development system. Each program or group of related programs is described in terms of a generalized invocation sequence and then a thorough explanation of functions and options.

In each synopsis description, arguments to a program are given in square brackets if they are optional. If a string of letters appear which do not apparently represent an English word, they give the list of available option switches. For example, the synopsis for the `ls` command reads:

SYNOPSIS

ls [-1ACFLRacdfgilnqrstu] *file...*

All of the characters “**1**”, “**A**”, “**C**”, and so on, are available as options to the **ls** program. If an argument is followed by an ellipsis, then any number of arguments of that form may appear.

The executable files containing the utility programs are almost without exception located in `/bin`.

NAME

`a2ps` – change an ascii file to postscript format

SYNOPSIS

a2ps [-#*num*] [[-?] [-b] [-f] [-f*num*] [[-h] [-i] [-m{lrtb}*num*] [-n] [-n{bfhinprsvw}] [-p] [-r] [-tnum] [-v] [-w] [-A4] [*file*] ...

DESCRIPTION

`a2ps` formats each named file for printing in a postscript printer; if no file is given, `a2ps` reads from the standard input. The format used is nice and compact: normally two pages on each physical page, borders surrounding pages, headers with useful information (page number, printing date, file name), line numbering, etc.

OPTIONS

-#*num* To get multiple copies of output, where *num* is the number of copies desired of each file named. By default, you get only one copy.

-? Print usage information.

-b Force printing binary files. By default, binary files printing is stopped before second page (see **-nb** option).

-f Fold lines too large to be printed inside the borders (default option). Max size depends on format and font size used.

-f*num* Changes font size to this value. Default values for font size are 6.8 in landscape mode and 9.0 in portrait mode. Try other sizes if you think that these values are too small (or big).

-h Print usage information.

-i Interpret `TAB`, `BS` and `FF` characters (default option). `TAB` is replaced by enough spaces to reach next tab stop while `BS` and `FF` have their meanings.

-n	Output lines are preceded by line numbers, numbered sequentially from `1` (default option).
-nb	Don't print binary files. To detect such a file we make use of a very simple heuristic: if the first page of the file contains at less 75% of non-printing characters, it's a binary file. First page is always printed.
-nf	Cut lines too large (don't fold).
-ml *num*	Set left margin, *num* is a floating point number.
-mr *num*	Set right margin, *num* is a floating point number.
-mt *num*	Set top margin, *num* is a floating point number.
-mb *num*	Set bottom margin, *num* is a floating point number.
-nh	Don't print page headers.
-ni	Don't interpret `TAB`, `BS`, and `FF` characters. They will be printed according to **-v** option.
-nn	Don't number output lines.
-np	Don't print pages in portrait mode (see **-p** option) but in landscape mode.
-nr	Sheet numbering (see **-r** option) must be continue for all files (don't reset on new file).
-ns	Don't print page surrounding borders.
-nv	Replace non-printing characters by a space.
-nw	Don't use wide format (see **-w** option).
-p	Print files in portrait mode, one page per physical sheet. The default option is to use landscape mode and print two pages on a physical sheet. This option may not be used with the **-w** option.
-r	Reset sheet numbering for each new file (default option). Sheet numbering is used to number physical pages (sheets printed) and is placed in the bottom of each physical page. It differs from page numbering: logical pages of file been printed.

-t*num* Set `TAB` size to *num* (default 8). This option is ignored if **-ni** is specified (see below).

-v Replace non-printing characters so that they are visible and easy to identify (default option). Control characters (ASCII codes lower than 0x20) are printed like `^X` for `Ctrl-x`; the delete character (hex 0x3f) is printed as `^?`. Non-ASCII characters (with the high bit set) are printed as `M-` (for meta) followed by the character of the low 7 bits. `TAB`, `BS`, and `FF` are handled like non-printing characters if **-ni** option is specified.

-w Use wide format pages. This will result in only one page per physical sheet but in landscape mode. This option may not be used with the **-p** option.

-A4 Print for A4 (European Standard) paper.

USAGE

`a2ps` sends formatted file to standard output. The user could redirect this output to a file, pipe it directly to a print command (like `lpr`), or pipe it to another filter for further processing.

```
a2ps file1 > file2
a2ps file1 | lpr -l -Pps
a2ps -p file1 | psnup -n4 | lpr -l -Pps
```

AUTHORS

Evan Kirshenbaum (`evan@csli`) for the initial version.

Miguel Santana (`miguel@imag.imag.fr`) for versions 1.0, 2.0, and 3.0.

SEE ALSO

Utility Programs - `lpr`

NAME

`apropos` – search the whatis database for strings

SYNOPSIS

`apropos` *keyword* ...

DESCRIPTION

`apropos` searches a set of database files containing short descriptions of system commands for keywords and displays the result on the standard output.

SEE ALSO

Utility Programs - `whatis, man`

NAME

`ar` – create, modify, and extract from archives.

SYNOPSIS

ar [-]{dmpqrtx}[abcilosuvV] [*membername***]** *archive files*...

DESCRIPTION

The GNU `ar` program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and may be reconstituted on extraction.

GNU `ar` can maintain archives whose members have names of any length; however, depending on how `ar` is configured on your system, a limit on member-name length may be imposed (for compatibility with archive formats maintained with other tools). If it exists, the limit is often 15 characters (typical of formats related to a.out) or 16 characters (typical of formats related to COFF).

`ar` is considered a binary utility because archives of this sort are most often used as *libraries* holding commonly needed subroutines.

`ar` will create an index to the symbols defined in relocatable object modules in the archive when you specify the modifier '`s`'. Once created, this index is updated in the archive whenever `ar` makes a change to its contents (save for the '`q`' update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use '`nm -s`' or '`nm --print-armap`' to list this index table. If an archive lacks the table, another form of `ar` called `ranlib` can be used to add just the table.

`ar` insists on at least two arguments to execute: one keyletter specifying the *operation* (optionally accompanied by other keyletters specifying *modifiers*), and the archive name to act on.

Most operations can also accept further *files* arguments, specifying particular files to operate on.

OPTIONS

GNU `ar` allows you to mix the operation code letter and several modifier flags in any order, within the first command-line argument.

If you wish, you may begin the first command-line argument with a dash.

The operation code letter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

`d`
: *Delete* modules from the archive. Specify the names of modules to be deleted as *files*; the archive is untouched if you specify no files to delete.

  If you specify the '**`v`**' modifier, `ar` will list each module as it is deleted.

`m`
: Use this operation to *move* members in an archive.

  The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

  If no modifiers are used with `m`, any members you name in the *files* arguments are moved to the *end* of the archive; you can use the '**`a`**', '**`b`**', or '**`i`**' modifiers to move them to a specified place instead.

`p`
: *Print* the specified members of the archive, to the standard output file. If the '**`v`**' modifier is specified, show the member name before copying its contents to standard output.

  If you specify no *files*, all the files in the archive are printed.

q	*Quick append*; add *files* to the end of *archive*, without checking for replacement.
	The modifiers '**a**', '**b**', and '**i**' do *not* affect this operation; new members are always placed at the end of the archive.
	The modifier '**v**' makes `ar` list each file as it is appended.
	Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use '`ar s`' or `ranlib` explicitly to update the symbol table index.
r	Insert *files* into *archive* (with *replacement*). This operation differs from '**q**' in that any previously existing members are deleted if their names match those being added.
	If one of the files named in *files* doesn't exist, `ar` displays an error message, and leaves undisturbed any existing members of the archive matching that name.
	By default, new members are added at the end of the file; but you may use one of the modifiers '**a**', '**b**', or '**i**' to request placement relative to some existing member.
	The modifier '**v**' used with this operation elicits a line of output for each file inserted, along with one of the letters '**a**' or '**r**' to indicate whether the file was appended (no old member deleted) or replaced.
t	Display a *table* listing the contents of *archive*, or those of the files listed in *files* that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the '**v**' modifier.
	If you do not specify any *files*, all files in the archive are listed.

If there is more than one file with the same name (say, '`fie`') in an archive (say '`b.a`'), '`ar t b.a fie`' will list only the first instance; to see them all, you must ask for a complete listing—in our example, '`ar t b.a`'.

x *Extract* members (named *files*) from the archive. You can use the '**v**' modifier with this operation, to request that `ar` list each name as it extracts it.

If you do not specify any *files*, all files in the archive are extracted.

A number of modifier code letters may immediately follow the operation code, to specify variations on an operation's behavior:

a Add new files *after* an existing member of the archive. If you use the modifier **a**, the name of an existing archive member must be present as the *membername* argument, before the *archive* specification.

b Add new files *before* an existing member of the archive. If you use the modifier **b**, the name of an existing archive member must be present as the *membername* argument, before the *archive* specification. (same as '**i**').

c *Create* the archive. The specified *archive* is always created if it didn't exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

f Truncate names in the archive. `ar` will normally permit file names of any length. This will cause it to create archives which are not compatible with the native `ar` program on some systems. If this is a concern, the **f** modifier may be used to truncate file names when putting them in the archive.

i Insert new files *before* an existing member of the archive. If you use the modifier **i**, the name of an existing archive member must be present as the *membername* argument, before the *archive* specification. (same as '**b**').

l This modifier is accepted but not used.

o	Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive will be stamped with the time of extraction.
s	Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running '`ar s`' on an archive is equivalent to running '`ranlib`' on it.
u	Normally, `ar r`... inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The '**u**' modifier is allowed only for the operation '**r**' (replace). In particular, the combination '`qu`' is not allowed, since checking the timestamps would lose any speed advantage from the operation '**q**'.
v	This modifier requests the *verbose* version of an operation. Many operations display additional information, such as filenames processed, when the modifier '**v**' is appended.
V	This modifier shows the version number of `ar`.

SEE ALSO

Utility Programs - `nm`, `ranlib`

Documents - *The GNU Binary Utilities*, Roland H. Pesch.

COPYING

NAME

`arp` – address resolution display and control

SYNOPSIS

arp *hostname*

arp -a

arp -d *hostname*

arp -s *hostname ether_addr* **[***temp***] [***pub***] [***trail***]**

arp -f *filename*

DESCRIPTION

The `arp` program displays and modifies the Internet-to-Ethernet address translation tables used by the address resolution protocol. With no flags, the program displays the current ARP entry for hostname. The host may be specified by name or by number, using Internet dot notation.

OPTIONS

Available options:

-a The program displays all of the current ARP entries.

-d A super-user may delete an entry for the host called hostname with the **-d** flag.

-s *hostname ether_addr*
Create an ARP entry for the host called *hostname* with the Ethernet address *ether_addr*. The Ethernet address is given as six hex bytes separated by colons. The entry will be permanent unless the word temp is given in the command. If the word pub is given, the entry will be "published"; i.e., this system will act as an ARP server, responding to requests for hostname even though the host address is not its own. The word trail indicates that trailer encapsulations may be sent to this host.

-f *filename* Causes the file *filename* to be read and multiple entries to be set in the ARP tables. Entries in the file should be of the form

hostname ether_addr [*temp*] [*pub*] [*trail*]
with argument meanings as given above.

SEE ALSO

Utility Programs - `ifconfig`

Library Functions - `inet`

HISTORY

The `arp` command appeared in 4.3BSD.

NAME

`as` – the portable GNU assembler.

SYNOPSIS

as [-a[dhlns] [=*file*]] [-D] [--defsym *SYM*=*VAL*] [-f]
[-I *path*] [-K] [-L] [-M | --mri] [-o *objfile*] [-R]
[-v] [-w] [-- | *files...*]

68k-only options:
[-l] [-mc68000|-mc68010|-mc68020] [-mc68030] [-mc68040]

DESCRIPTION

GNU `as` is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops)* and assembler syntax.

For information on the syntax and pseudo-ops used by GNU `as`, see the manual *Using as: The GNU Assembler*).

`as` is primarily intended to assemble the output of the GNU C compiler `gcc` for use by the linker `ld`. Nevertheless, we've tried to make `as` assemble correctly everything that the native assembler would. This doesn't mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Each time you run `as` it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

If `as` is given no file names it attempts to read one input file from the `as` standard input, which is normally your terminal. You may have to type `ctl-D` to tell `as` there is no more program to assemble. Use '**--**' if you need to explicitly name the standard input file in your command line.

`as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when `as` is run automatically by a compiler. Warnings report an assumption made so that `as` could keep assembling a flawed program; errors report a grave problem that stops the assembly.

OPTIONS

-a Turn on assembly listings. There are various suboptions. `d` omits debugging directives. `h` includes the high level source code; this is only available if the source file can be found, and the code was compiled with **-g**. `l` includes an assembly listing. `n` omits forms processing. `s` includes a symbol listing. = *file* sets the listing file name; this must be the last suboption. The default suboptions are `hls`.

-D This option is accepted only for script compatibility with calls to other assemblers; it has no effect on `as`.

--defsym *SYM*=*VALUE*
Define the symbol *SYM* to be *VALUE* before assembling the input file. *VALUE* must be an integer constant. As in C, a leading 0x indicates a hexadecimal value, and a leading 0 indicates an octal value.

-f Attain *faster* performance by skipping preprocessing (assume source is compiler output).

-I *path* Add *path* to the search list for `.include` directives.

-K Issue warnings when difference tables altered for long displacements.

-L Keep (in symbol table) local symbols, starting with ‘`L`’

-M, --mri Assemble in MRI compatibility mode.

-o *objfile* Name the object-file output from `as`

-R Fold data section into text section

-v Announce `as` version

-W Suppress warning messages

- - | *files...*	Source files to assemble, or standard input (**--**)
-l	*(When configured for Motorola 68000).* Shorten references to undefined symbols, to one word instead of two.

-mc68000|-mc68010|-mc68020|-mc68030|-mc68040
(When configured for Motorola 68000). Specify what processor in the 68000 family is the target.

Options may be in any order, and may be before, after, or between file names. The order of file names is significant.

'**--**' (two hyphens) by itself names the standard input file explicitly, as one of the files for `as` to assemble.

Except for '**--**' any command line argument that begins with a hyphen ('**-**') is an option. Each option changes the behavior of `as`. No option changes the way another option works. An option is a '**-**' followed by one or more letters; the case of the letter is important. All options are optional.

The '**-o**' option expects exactly one file name to follow. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard).

These two command lines are equivalent:

```
as  -o  my-object-file.o  mumble.s
as  -omy-object-file.o  mumble.s
```

SEE ALSO

Utility Programs - `gcc`, `ld`

Documents - *Using as: The GNU Assembler*

COPYING

NAME

`atob, btoa` – encode/decode binary to printable ASCII

SYNOPSIS

`btoa`
`atob`

DESCRIPTION

`btoa` is a filter that reads anything from the standard input and encodes it into printable ASCII on the standard output. It also attaches a header and checksum information used by the reverse filter `atob` to find the start of the data and to check integrity.

`atob` reads an encoded file, strips off any leading and trailing lines added by mailers, and recreates a copy of the original file on the standard output. `atob` gives NO output (and exits with an error message) if its input is garbage or the checksums do not check.

FEATURES

`btoa` uses a compact base-85 encoding so that 4 bytes are encoded into 5 characters (file is expanded by 25%). As a special case, 32-bit zero is encoded as one character. This encoding produces less output than `uuencode`.

COPYRIGHT NOTICE

The `atob` and `btoa` utilities were written by Mr. Paul Rutter and Mr. Joe Orost, not by Lynx Real-Time Systems. The source code is apparently intended to be freely distributable, and is thus available upon request.

SEE ALSO

Utility Programs - `compress, emit`

NAME

`autoexpect` – generate an Expect script from watching a session

SYNOPSIS

autoexpect [*args* **] [** *program args...* **]**

DESCRIPTION

`autoexpect` watches you interacting with another program and creates an Expect script that reproduces your interactions. For straightline scripts, `autoexpect` saves substantial time over writing scripts by hand. Even if you are an Expect expert, you will find it convenient to use `autoexpect` to automate the more mindless parts of interactions. It is much easier to cut/paste hunks of `autoexpect` scripts together than to write them from scratch. And if you are a beginner, you may be able to get away with learning nothing more about Expect than how to call `autoexpect`.

The simplest way to use `autoexpect` is to call it from the command line with no arguments. For example:

```
% autoexpect
```

By default, `autoexpect` spawns a shell for you. Given a program name and arguments, `autoexpect` spawns that program. For example:

```
% autoexpect ftp ftp.cme.nist.gov
```

Once your spawned program is running, interact normally. When you have exited the shell (or program that you specified), `autoexpect` will create a new script for you. By default, `autoexpect` writes the new script to "`script.exp`". You can override this with the **-f** flag followed by a new script name.

The following example runs "`ftp ftp.cme.nist.gov`" and stores the resulting Expect script in the file "`nist`".

```
% autoexpect -f nist ftp ftp.cme.nist.gov
```

It is important to understand that `autoexpect` does not guarantee a working script because it necessarily has to guess about certain things - and occasionally it guesses wrong. However, it is usually very easy to identify and fix these problems. The typical problems are:

- Timing. A surprisingly large number of programs (`rn`, `ksh`, `zsh`, `telnet`, etc.) and devices (e.g., modems) ignore keystrokes that arrive "too quickly" after prompts. If you find your new script hanging up at one spot, try adding a short sleep just before the previous send.

  The **-c** flag puts `autoexpect` into "conservative mode". This mode makes `autoexpect` automatically pause briefly (a tenth of a second) before sending each character. This pacifies every program I know of.

  Fortunately, these spots are rare. For example, telnet ignores characters only after entering its escape sequence. Modems only ignore characters immediately after connecting to them for the first time. A few programs exhibit this behavior all the time but typically have a switch to disable it. For example, the **-T** flag of `rn` disables this behavior.

  The following example starts `autoexpect` in conservative mode.

  autoexpect -c

  The **-C** flag defines a key to toggle conservative mode. The following example starts `autoexpect` (in non-conservative mode) with ^L as the toggle. (Note that the ^L is entered literally.)

  autoexpect -C ^L

  The following example starts `autoexpect` in conservative mode with ^L as the toggle.

  autoexpect -c -C ^L

  Echoing. Many program echo characters. For example, if you type "more" to a shell, what `autoexpect` actually sees is:

  ```
  you typed 'm',
  computer typed 'm',
  you typed 'o',
  computer typed 'o',
  ```

```
you typed 'r',
computer typed 'r',
...
```

Without specific knowledge of the program, it is impossible to know if you are waiting to see each character echoed before typing the next. If `autoexpect` sees characters being echoed, it assumes that it can send them all as a group rather than interleaving them the way they originally appeared. This makes the script more pleasant to read. However, it could conceivably be incorrect if you really had to wait to see each character echoed.

- Change. `autoexpect` records every character from the interaction in the script. This is desirable because it gives you the ability to make judgements about what is important and what can be replaced with a pattern match.

  On the other hand, if you use commands whose output differs from run to run, the generated scripts are not going to be correct. For example, the "`date`" command always produces different output. So using the date command while running `autoexpect` is a sure way to produce a script that will require editing in order for it to work.

  The **-p** flag puts `autoexpect` into "prompt mode". In this mode, `autoexpect` will only look for the last line of program output, which is usually the prompt. This handles the date problem (see above) and most others.

  The following example starts `autoexpect` in prompt mode.

  `autoexpect -p`

  The **-P** flag defines a key to toggle prompt mode. The following example starts `autoexpect` (in non-prompt mode) with ^P as the toggle. Note that the ^P is entered literally.

  `autoexpect -P ^P`

  The following example starts `autoexpect` in prompt mode with ^P as the toggle.

  `autoexpect -p ^P`

OPTIONS

The **-quiet** flag disables informational messages produced by `autoexpect`.

The **-Q** flag names a quote character which can be used to enter characters that `autoexpect` would otherwise consume because they are used as toggles.

The following example shows a number of flags with quote used to provide a way of entering the toggles literally.

```
autoexpect -P ^P -C ^L -Q ^Q
```

STYLE

I don't know if there is a "style" for Expect programs but `autoexpect` should definitely not be held up as any model of style. For example, `autoexpect` uses features of Expect that are intended specifically for computer-generated scripting. So don't try to faithfully write scripts that appear as if they were generated by `autoexpect`. This is not useful.

On the other hand, `autoexpect` scripts do show some worthwhile things. For example, you can see how any string must be quoted in order to use it in a Tcl script simply by running the strings through `autoexpect`.

SEE ALSO

Documents - *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs* by Don Libes.

AUTHOR

Don Libes, National Institute of Standards and Technology

`expect` and `autoexpect` are in the public domain. NIST and I would appreciate credit if these programs or parts of them are used.

NAME

banner – show arguments in big letters

SYNOPSIS

banner [*string*] ...

DESCRIPTION

banner prints each *string* in big (88) letters on standard output. Each *string* is truncated to ten characters. Separate *strings* are printed on separate "lines".

Lower-case characters are converted to upper-case.

SEE ALSO

Utility Programs - echo

NAME

`basename, dirname` – return portions of path names

SYNOPSIS

`basename` *path* [*suffix*]
`dirname` *path*

DESCRIPTION

`basename` deletes the given *suffix* and any prefix ending in `/` from *path*, and prints the result on the standard output.

`dirname` deletes any suffix beginning with `/` from *path*, leaving the directory portion of the path name, and prints the result on the standard output.

These utilities are typically used in shell scripts.

SEE ALSO

Utility Programs - `sh`

NAME

bash – GNU Bourne-Again SHell

SYNOPSIS

bash [*options***] [***file***]**

DESCRIPTION

bash is an sh-compatible command language interpreter that executes commands read from the standard input or from a file. bash also incorporates useful features from the Korn and C shells (ksh and csh).

bash is ultimately intended to be a faithful implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2).

OPTIONS

In addition to the single-character shell options documented in the description of the set built-in command, bash interprets the following flags when it is invoked:

–c *string* If the **–c** flag is present, then commands are read from *string*.

–i If the **–i** flag is present, the shell is *interactive*.

–s If the **–s** flag is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell.

– A single **–** signals the end of options and disables further option processing. Any arguments after the **–** are treated as filenames and arguments. An argument of **––** is equivalent to an argument of **–**.

`bash` also interprets a number of multi-character options. These options must appear on the command line before the single-character options to be recognized.

-norc Do not load the personal initialization file `~/.bashrc` if the shell is interactive. This is the default if the shell name is `sh`.

-noprofile Do not read either `/etc/profile` or `~/.bash_profile`. By default, `bash` normally reads these files when it is invoked as a login shell.

-rcfile *file*

Execute commands from *file* instead of the standard personal initialization file `~/.bashrc`, if the shell is interactive.

-version Show the version number of this instance of `bash` when starting.

-quiet Do not be verbose when starting up (do not show the shell version or any other information).

-login Make `bash` act as if it had been invoked by `login`.

-nobraceexpansion

Do not perform curly brace expansion as with `csh`.

-nolineediting

Do not use the GNU `readline` library to read command lines if interactive.

ARGUMENTS

If arguments remain after option processing, and neither the **-c** nor the **-s** option has been supplied, the first argument is assumed to be the name of a file containing shell commands. If `bash` is invoked in this fashion, `$0` is set to the name of the file, and the positional parameters are set to the remaining arguments. `bash` reads and executes commands from this file, then exits.

DEFINITIONS

`blank` A space or tab.

`word` A sequence of characters considered as a single unit by the shell. Also known as a `token`.

`name` A *word* consisting only of alphanumeric characters and underscores, and beginning with an alphabetic character or an underscore. Also referred to as an `identifier`.

`metacharacter`
A character that, when unquoted, separates words. One of the following: `| & ; ( ) < > <space>`

`control operator`
A token that performs a control function. It is one of the following symbols: `|| & && ; ;; ( ) |`

RESERVED WORDS

Reserved words are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a simple command (See “SHELL GRAMMAR” for more information.) or the third word of a `case` or `for` command:

```
! case do done elif else esac fi
```

SHELL GRAMMAR

Simple Commands

A simple command is a sequence of optional variable assignments followed by blank-separated words and redirections, and terminated by a control operator. The first word specifies the command to be executed. The remaining words are passed as arguments to the invoked command.

The return value of a simple command is its exit status, or 128+*n* if the command is terminated by signal *n*.

Pipelines

A pipeline is a sequence of one or more commands separated by the character |. The format for a pipeline is:

`[!]` *command* `[|` *command2 ...*`]`

The standard output of *command* is connected to the standard input of *command2*. This connection is performed before any redirections specified by the command (See "REDIRECTION" for more information.).

If the reserved word `!` precedes a pipeline, the exit status of that pipeline is the logical NOT of the exit status of the last command. Otherwise, the status of the pipeline is the exit status of the last command. The shell waits for all commands in the pipeline to terminate before returning a value.

Each command in a pipeline is executed as a separate process (i.e. in a subshell).

Lists

A *list* is a sequence of one or more pipelines separated by one of the operators `;`, `&`, `&&`, or `||`, and optionally terminated by one of `;`, `&`, or `<newline>`.

Of these list operators, `&&` has highest precedence; `||` has the next highest precedence, followed by `;` and `&`, which have equal precedence.

If a command is terminated by the control operator `&`, the shell executes the command in the background in a subshell. The shell does not wait for the command to finish. Commands separated by a `;` are executed sequentially; the shell waits for each command to terminate in turn.

The control operators `&&` and `||` denote AND lists and OR lists, respectively. An AND list has the form

command `&&` *command2*

command2 is executed if, and only if, *command* returns an exit status of zero.

An OR list has the form

command `||` *command2*

command2 is executed if and only if *command* returns a non-zero exit status.

Compound Commands

A compound command is one of the following:

(*list*) *list* is executed in a subshell. Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes.

{ *list*; } *list* is simply executed in the current shell environment. This is known as a group command.

`for` *name* [`in` *word*;] `do` *list* ; `done`
The list of words following `in` is expanded, generating a list of items. The variable *name* is set to each element of this list in turn, and *list* is executed each time. If the `in` *word* is omitted, the `for` command executes *list* once for each positional parameter that is set. (See "PARAMETERS" for more information.) The exit status is the exit status of the last command, or zero if no commands were executed.

`case` *word* `in` [*pattern* [| *pattern*] ...) *list* ;;] ... `esac`
A `case` command first expands *word*, and tries to match it against each *pattern* in turn. When a match is found, the corresponding *list* is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no patterns are matches. Otherwise, it is the exit status of the last command executed in *list*.

`if` *list* `then` *list* [`elif` *list* `then` *list*] ... [`else` *list*] `fi`
The `if` *list* is executed. If its exit status is zero, the `then` *list* is executed. Otherwise, each `elif` *list* is executed in turn, and if its exit status is zero, the corresponding `then` *list* is executed and the command completes. Otherwise, the `else` *list* is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.

`while` *list* `do` *list* `done`

`until` *list* `do` *list* `done`
The `while` command continuously executes the `do` *list* as long as the last command in *list* returns an exit status of zero. The `until` command is identical to the

`while` command, except that the test is negated; the `do` *list* is executed as long as the last command in *list* returns a non-zero exit status. The exit status of the `while` and `until` commands is the exit status of the last `do` *list* command executed, or zero if none was executed.

[`function`] *name*`()` { *list*; }
This defines a function named *name*. The body of the function is the *list* of commands between { and }. This list is executed whenever *name* is specified as the name of a simple command. The exit status of a function is the exit status of the last command executed in the body. (See “FUNCTIONS” for more information.)

COMMENTS

In a non-interactive shell, a word beginning with `#` causes that word and all remaining characters on that line to be ignored.

QUOTING

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the metacharacters listed under DEFINITIONS has special meaning to the shell and must be quoted if they are to represent themselves. There are three quoting mechanisms: the escape character, single quotes, and double quotes.

A non-quoted backslash (\) is the escape character. It preserves the literal value of the next character that follows, with the exception of `<newline>`. If a `\<newline>` pair appears, it is treated as a line continuation (that is, it is effectively ignored), if the backslash is non-quoted.

Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of `$`, `` ` ``, and `\`. The characters `$` and `` ` `` retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: `$`, `` ` ``, `"`, `\`, or `<newline>`. A double quote may be quoted within double quotes by preceding it with a backslash.

The special parameters `*` and `@` have special meaning when in double quotes.(See “PARAMETERS” for more information.)

PARAMETERS

A parameter is an entity that stores values, somewhat like a variable in a conventional programming language. It can be a *name*, a number, or one of the special characters listed under “Special Parameters”. For the shell’s purposes, a variable is a parameter denoted by a *name*.

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the `unset` builtin command.

A variable may be assigned to by a statement of the form

name=[*value*]

If *value* is not given, the variable is assigned the null string. All *values* undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. If the variable has its **`-i`** attribute set (see `declare`) then *value* is subject to arithmetic expansion even if the `$[...]` syntax does not appear. Word splitting is not performed, with the exception of `$@` as explained under Special Parameters. Pathname expansion is not performed.

Positional Parameters

A positional parameter is a parameter denoted by one or more digits, other than the single digit `0`. Positional parameters are assigned from the shell’s arguments when it is invoked, and may be reassigned using the `set` builtin command. The positional parameters are temporarily replaced when a shell function is executed. (See “FUNCTIONS” for more information.)

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.(See “EXPANSION” for more information.)

Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

`*` Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable. That is, `$*` is equivalent to `$1`*c*`$2`*c*`...`, where *c* is the first character of the value of the `IFS` variable. If `IFS` is null or unset, the parameters are separated by spaces.

`@` Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands as a separate word. That is, `$@` is equivalent to `$1 $2` ... When there are no positional parameters, `$@` and `$@` expand to nothing (i.e. they are removed).

`#` Expands to the number of positional parameters in decimal.

`?` Expands to the status of the most recently executed foreground pipeline.

`–` Expands to the current option flags as specified upon invocation, by the `set` builtin command, or those set by the shell itself (such as the **`-i`** flag).

`$` Expands to the process ID of the shell. In a `()` subshell, it expands to the process ID of the current shell, not the subshell.

`!` Expands to the process ID of the most recently executed background (asynchronous) command.

`0`	Expands to the name of the shell or shell script. This is set at shell initialization. If `bash` is invoked with a file of commands, `$0` is set to the name of that file. Otherwise, it is set to the pathname used to invoke `bash`, as given by argument zero.
`_`	Expands to the last argument to the previous command, after expansion. Also set to the full pathname of each command executed and placed in the environment exported to that command.

Shell Variables

The following variables are set by the shell:

`PPID`	The process ID of the shell's parent.
`PWD`	The current working directory as set by the `cd` command.
`OLDPWD`	The previous working directory as set by the `cd` command.
`REPLY`	Set to the line of input read by the `read` builtin command when no arguments are supplied.
`UID`	Expands to the user ID of the current user.
`EUID`	Expands to the effective user ID of the current user.
`BASH`	Expands to the full pathname used to invoke this instance of `bash`.
`BASH_VERSION`	Expands to the version number of this instance of `bash`.
`SHLVL`	Incremented by one each time an instance of `bash` is started.
`RANDOM`	Each time this parameter is referenced, a random integer is generated. The sequence of random numbers may be initialized by assigning a value to `RANDOM`. If `RANDOM` is unset, it loses its special properties, even if it is subsequently reset.

`SECONDS`
: Each time this parameter is referenced, the number of seconds since shell invocation is returned. If a value is assigned to `SECONDS`, the value returned upon subsequent references is the number of seconds since the assignment plus the value assigned. If `SECONDS` is unset, it loses its special properties, even if it is subsequently reset.

`LINENO`
: Each time this parameter is referenced, the shell substitutes a decimal number representing the current sequential line number (starting with 1) within a script or function. When not in a script or function, the value substituted is not guaranteed to be meaningful. When in a function, the value is not the number of the source line that the command appears on (that information has been lost by the time the function is executed), but is an approximation of the number of simple commands executed in the current function. If `LINENO` is unset, it loses its special properties, even if it is subsequently reset.

`OPTARG`
: The value of the last option argument processed by the `getopts` builtin command.

`OPTIND`
: The index of the last option processed by the `getopts` builtin command.

The following variables are used by the shell. In some cases, `bash` assigns a default value to a variable; these cases are noted below.

`IFS`
: The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the `read` builtin command. The default value is <space><tab><newline>.

`PATH`
: The search path for commands. It is a colon-separated list of directories in which the shell looks for commands. (See “COMMAND EXECUTION” for more information.) The default path is system-dependent, and is set by the administrator who installs `bash`. A common value is:

```
.:/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:
/usr/bin: /etc:/usr/etc
```

Note that in some circumstances a leading `.` in `PATH` can be a security hazard.

`HOME`
: The home directory of the current user; the default argument for the `cd` builtin command.

`CDPATH`
: The search path for the `cd` command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the `cd` command. A sample value is `.:~:/usr`.

`ENV`
: If this parameter is set when `bash` is executing a shell script, its value is interpreted as a filename containing commands to initialize the shell, as in `.bashrc`. The value of `ENV` is subjected to parameter expansion, command substitution, and arithmetic expansion before being interpreted as a pathname. `PATH` is not used to search for the resultant pathname.

`MAIL`
: If this parameter is set to a filename and the `MAILPATH` variable is not set, `bash` informs the user of the arrival of mail in the specified file.

`MAILCHECK`
: Specifies how often (in seconds) `bash` checks for mail. The default is 60 seconds. When it is time to check for mail, the shell does so before prompting. If this variable is unset, the shell disables mail checking.

`MAILPATH`
: A colon-separated list of pathnames to be checked for mail. The message to be printed may be specified by separating the pathname from the message with a `?`. `$_` stands for the name of the current `mailfile`. Example:

```
MAILPATH='/usr/spool/mail/bfox?"You have\
mail":~/shell-mail?"$_ has mail!"'
```

`bash` supplies a default value for this variable, but the location of the user mail files that it uses is system dependent (e.g., `/usr/spool/mail/$USER`).

`MAIL_WARNING`
: If set, and a file that `bash` is checking for mail has been accessed since the last time it was checked, the message `The mail in mailfile has been read` is printed.

`PS1`	The value of this parameter is expanded and used as the primary prompt string. (See "PROMPTING" for more information.) The default value is `bash\$` .
`PS2`	The value of this parameter is expanded like `PS1` and used as the secondary prompt string. The default is `>` .
`PS4`	The value of this parameter is expanded like `PS1` and the value is printed before each command `bash` displays during an execution trace. The first character of `PS4` is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is `+`.
`NO_PROMPT_VARS`	If set, the decoded prompt string does not undergo further expansion. (See "PROMPTING" for more information.)
`HISTSIZE`	The number of commands to remember in the command history. (See "HISTORY" for more information.)
`HISTFILE`	The name of the file in which command history is saved. (See "HISTORY" for more information.)
`HISTFILESIZE`	The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines.
`OPTERR`	If set to the value 1, `bash` displays error messages generated by the `getopts` builtin command. `OPTERR` is initialized to 1 each time the shell is invoked or a shell script is executed.
`PROMPT_COMMAND`	If set, the value is executed as a command prior to issuing each primary prompt.
`IGNOREEOF`	
`ignoreeof`	Controls the action of the shell on receipt of an `EOF` character as the sole input. If set, the value is the number of consecutive `EOF` characters typed before `bash` exits. If the variable exists but does not have a

numeric value, or has no value, the default value is 10. If it does not exist, EOF signifies the end of input to the shell. This is only in effect for interactive shells.

HOSTTYPE
: Automatically set to a string that uniquely describes the type of machine on which bash is executing. The default is system-dependent.

TMOUT
: If set to a value greater than zero, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt. bash terminates after waiting for that number of seconds if input does not arrive.

FCEDIT
: The default editor for the fc builtin command.

FIGNORE
: A colon-separated list of suffixes to ignore when performing filename completion. (See “READLINE” for more information.) A filename whose suffix matches one of the entries in FIGNORE is excluded from the list of matched filenames. A sample value is .o:~.

notify
: If set, bash reports terminated background jobs immediately, rather than waiting until before printing the next primary prompt.

history_control
: If set to a value of ignorespace, it means don’t enter lines which begin with a <space> on the history list. If set to a value of ignoredups, it means don’t enter lines which match the last entered line. If unset, or if set to any other value than those above, all lines read by the parser are saved on the history list.

command_oriented_history
: If set, bash attempts to save all lines of a multiple-line command in the same history entry. This allows easy re-editing of multi-line commands.

glob_dot_filenames
: If set, bash includes filenames beginning with a . (period) in the results of pathname expansion.

`allow_null_glob_expansion`
: If set, `bash` allows pathname patterns which match no files (See “Pathname Expansion” for more information.) to expand to a null string, rather than themselves.

`histchars`
: The two characters which control history expansion and tokenization. The first character is the history expansion character, that is, the character which signals the start of a history expansion, normally `!`. The second character is the character which signifies that the remainder of the line is a comment, when found as the first character of a word.

`nolinks`
: If set, the shell does not follow symbolic links when executing commands that change the current working directory. It uses the physical directory structure instead. By default, `bash` follows the logical chain of directories when performing commands such as `cd`.

`hostname_completion_file`
: Contains the name of a file in the same format as `/etc/hosts` that should be read when the shell needs to complete a hostname. You can change the file interactively; the next time you want to complete a hostname `bash` adds the contents of the new file to the already existing database.

`noclobber`
: If set, `bash` does not overwrite an existing file with the `>`, `>&`, and `<>` redirection operators. This variable may be overridden when creating output files by using the redirection operator `>|` instead of `>` (see also the **-C** option to the `set` builtin command).

`auto_resume`
: This variable controls how the shell interacts with the user and job control. If this variable is set, single word simple commands without redirections are treated as candidates for resumption of an existing stopped job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, the job most recently accessed is selected.

`no_exit_on_failed_exec`
: If this variable exists, the shell does not exit if it cannot execute the file specified in the `exec` command.

`cdable_vars`	If this is set, an argument to the `cd` builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.
`pushd_silent`	If set, the `pushd` and `popd` builtin commands do not print the current directory stack after successful execution.

EXPANSION

Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, word splitting, and pathname expansion.

The order of expansions is: brace expansion, tilde expansion, parameter, variable, command, and arithmetic substitution (done in a left-to-right fashion), word splitting, and pathname expansion.

Only brace expansion, word splitting, and pathname expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The single exception to this is the expansion of `$@` as explained under “PARAMETERS”.

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to pathname expansion, but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional preamble, followed by a series of comma-separated strings between a pair of braces, followed by an optional postamble. The preamble is prepended to each string contained within the braces, and the postamble is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example, `a{d,c,b}e` expands into `ade ace abe`.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. `bash` does not apply any syntactic interpretation to the context of the expansion or the text between the braces.

This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example:

```
mkdir /usr/local/src/bash/{old,new,dist,bugs}
```

or

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

Brace expansion introduces a slight incompatibility with traditional versions of `sh`, the Bourne shell. `sh` does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. `bash` removes braces from words as a consequence of brace expansion. For example, a word entered to `sh` as *file{1,2}* appears identically in the output. The same word is output as *file1 file2* after expansion by `bash`. If strict compatibility with `sh` is desired, start `bash` with the **`-nobraceexpansion`** flag (See “OPTIONS” for more information.) or disable brace expansion with the `+o braceexpand` option to the `set` command. (See “SHELL BUILTIN COMMANDS” for more information.)

Tilde Expansion

If a word begins with a tilde character (~), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible login name. If this login name is the null string, the tilde is replaced with the value of the parameter `HOME`. If `HOME` is unset, the home directory of the user executing the shell is substituted instead.

If a + follows the tilde, the value of `PWD` is substituted. If a – follows, the value of `OLDPWD` is used.

Each variable assignment is checked for unquoted instances of tildes following a **:** or =. In these cases, tilde substitution is also performed. Consequently, one may use pathnames with tildes in `PATH`, `MAILPATH`, and `CDPATH`, and the shell will export the expanded variables.

Parameter Expansion

The `$` character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

${*parameter*}
: The value of *parameter* is substituted. The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character which is not to be interpreted as part of its name.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. `bash` tests for a parameter that is unset or null; omitting the colon results in a test only for a parameter that is unset.

`${`*parameter*`:-`*word*`}`
: Use default values. If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.

`${`*parameter*`:=`*word*`}`
: Assign default values. If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.

`${`*parameter*`:?`*word*`}`
: Display error if null or unset. If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted.

`${`*parameter*`:+`*word*`}`
: Use alternate value. If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.

`${#`*parameter*`}` The length in characters of the value of *parameter* is substituted. If *parameter* is `*` or `@`, the length substituted is the length of `*` expanded within double quotes.

`${`*parameter*`#`*word*`}`

`${`*parameter*`##`*word*`}`
The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches the beginning of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern deleted (the `#`case) or the longest matching pattern deleted (the `##` case).

`${`*parameter*`%`*word*`}`

`${`*parameter*`%%`*word*`}`
The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches a trailing portion of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern deleted (the `%` case) or the longest matching pattern deleted (the `%%` case).

Command Substitution

Command substitution allows the output of a command to replace the command name. There are two forms:

```
$(command)
```

or

```
`command`
```

`bash` performs the expansion by executing *command* and replacing the command substitution with the standard output of the command, with any trailing newlines deleted.

When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by $, `, or \. When using the $(command) form, all characters between the parentheses make up the command; none are treated specially.

Command substitutions may be nested. To nest when using the old form, escape the inner backquotes with backslashes.

If the substitution appears within double quotes, word splitting and pathname expansion are not performed on the results.

Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

$[*expression*]

The *expression* is treated as if it were within double quotes, but a double quote inside the braces is not treated specially. All tokens in the expression undergo parameter expansion, command substitution, and quote removal. Arithmetic substitutions may be nested.

The evaluation is performed according to the rules under "ARITHMETIC EVALUATION". If *expression* is invalid, bash prints a message indicating failure and no substitution occurs.

Word Splitting

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for *word splitting*.

The shell treats each character of IFS as a delimiter, and splits the results of the other expansions into words on these characters. If the value of IFS is exactly <space><tab><newline>, the default, then any sequence of IFS characters serves to delimit words; otherwise each occurrence of an IFS character is treated as a delimiter. If the value of IFS is null, no word splitting occurs. IFS cannot be unset.

Explicit null arguments (`""` or `''`) are retained. Implicit null arguments, resulting from the expansion of *parameters* that have no values, are removed.

Note that if no expansion occurs, no splitting is performed.

Pathname Expansion

After word splitting, `bash` scans each *word* for the characters `*`, `?`, and `[`, unless the **-f** flag has been set. If one of these characters appears, then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of pathnames matching the pattern. If no matching pathnames are found, and the shell variable `allow_null_glob_expansion` is unset, the word is left unchanged. If the variable is set, the word is removed if no matches are found. When a pattern is used for pathname generation, the character `.` at the start of a name or immediately following a slash must be matched explicitly, unless the shell variable `glob_dot_filenames` is set. The slash character must always be matched explicitly. In other cases, the `.` character is not treated specially.

The special pattern characters have the following meanings:

`*`	Matches any string, including the null string.
`?`	Matches any single character.
`[...]`	Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a *range*; any character lexically between those two characters, inclusive, is matched. If the first character following the `[` is a `!` or a `^` then any character not enclosed is matched. A – or `]` may be matched by including it as the first or last character in the set.

Quote Removal

After the preceding expansions, all unquoted occurrences of the characters `\`, `'`, and `"` are removed.

REDIRECTION

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. The following redirection operators may appear anywhere in a simple command or may precede or follow a *command*. Redirections are processed in the order they appear, from left to right.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is <, the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is >, the redirection refers to the standard output (file descriptor 1).

The word that follows the redirection operator in the following descriptions is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, and pathname expansion. If it expands to more than one word, `bash` reports an error.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor *n*, or the standard input (file descriptor `0`) if *n* is not specified.

The general format for redirecting input is:

[*n*]**<***word*

Redirecting Output

Redirection of output causes the file whose name results from the expansion of *word* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

The general format for redirecting output is:

[*n*]**>***word*

If the redirection operator is `>|`, then the variable `noclobber` is not consulted, and the file is created regardless of the value of `noclobber`. (See “Shell Variables” for more information.).

Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of *word* to be opened for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

The general format for appending output is:

`[`*n*`]>>`*word*

Redirecting Standard Output and Standard Error

`bash` allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word* with this construct.

There are two formats for redirecting standard output and standard error:

`&>`*word*

and

`>&`*word*

Of the two forms, the first is preferred. This is semantically equivalent to

`>`*word* `2>&1`

Here Documents

This type of redirection instructs the shell to read input from the current source until a line containing only *word* (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command.

The format of *here_documents* is as follows:

```
<<[-]word
        here_document
delimiter
```

No parameter expansion, command substitution, pathname expansion, or arithmetic expansion is performed on *word*. If any characters in *word* are quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. Otherwise, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion. In the latter case, the pair `\<newline>` is ignored, and `\` must be used to quote the characters `\`, `$`, and `` ` ``.

If the redirection operator is `<<-`, then all leading tab characters are stripped from input lines and the line containing *delimiter*. This allows here-documents within shell scripts to be indented in a natural fashion.

Duplicating File Descriptors

The redirection operator

```
[n]<&word
```

is used to duplicate input file descriptors. If *word* expands to one or more digits, the file descriptor denoted by *n* is made to be a copy of that file descriptor. If *word* evaluates to -, file descriptor *n* is closed. If *n* is not specified, the standard input (file descriptor 0) is used.

The operator

```
[n]>&word
```

is used similarly to duplicate output file descriptors. If *n* is not specified, the standard output (file descriptor 1) is used.

Opening File Descriptors for Reading and Writing

The redirection operator

```
[n]<>word
```

causes the file whose name is the expansion of *word* to be opened for both reading and writing on file descriptor *n*, or as the standard input and standard output if *n* is not specified.

Note that the order of redirections is significant. For example, the command

`ls >` *dirlist* `2>&1`

directs both standard output and standard error to the file *dirlist*, while the command

`ls 2>&1 >` *dirlist*

directs only the standard output to file *dirlist*, because the standard error was duplicated as standard output before the standard output was redirected to *dirlist*.

FUNCTIONS

A shell function, defined as described under SHELL GRAMMAR, stores a series of commands for later execution. However, functions are executed in the context of the current shell; no new process is created to interpret them (contrast this with the execution of a shell script). When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter `#` is updated to reflect the change. Positional parameter 0 is unchanged.

Variables local to the function may be declared with the `local` builtin command. Ordinarily, variables and their values are shared between the function and its caller.

If the builtin command `return` is executed in a function, the function completes and execution resumes with the next command after the function call. When a function completes, the values of the positional parameters and the special parameter `#` are restored to the values they had prior to function execution.

Function names may be listed with the **`-f`** option to the `declare` or `typeset` builtin commands. Functions may be exported so that subshells automatically have them defined with the **`-f`** option to the `export` builtin.

Functions may be recursive. No limit is imposed on the number of recursive calls.

ALIASES

The shell maintains a list of aliases that may be set and unset with the `alias` and `unalias` builtin commands. The first word of each command is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The `alias` name and the replacement text may contain any valid shell input, including the metacharacters listed above. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias `ls` to `ls-F`, for instance, and `bash` does not try to recursively expand the replacement text. If the last character of the alias value is a blank, then the next command word following the alias is also checked for alias expansion.

Aliases are created and listed with the `alias` command, and removed with the `unalias` command.

There is no mechanism for using arguments in the replacement text, as with `csh`. If arguments are needed, a shell function should be used.

The rules concerning the definition and use of aliases are somewhat confusing. `bash` always reads at least one complete line of input before executing any of the commands on that line. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. This means that the commands following the alias definition on that line are not affected by the new alias.

This behavior is also an issue when functions are executed. Aliases are expanded when the function definition is read, not when the function is executed, because a function definition is itself a compound command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use `alias` in compound commands.

Aliases are not expanded when the shell is not interactive.

Note that for almost every purpose, aliases are superseded by shell functions.

JOB CONTROL

Job control refers to the ability to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the system's terminal driver and `bash`.

The shell associates a job with each pipeline. It keeps a table of currently executing jobs, which may be listed with the `jobs` command. When `bash` starts a job asynchronously (in the *background*), it prints a line that looks like:

```
[1] 25647
```

indicating that this job is job number 1 and that the process ID of the single process in the job is 25647. `bash` uses the job abstraction as the basis for job control.

To facilitate the implementation of the user interface to job control, the system maintains the notion of a current terminal process group ID. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as `SIGINT`. These processes are said to be in the foreground.

Background processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or write to the terminal. Background processes which attempt to read from (write to) the terminal are sent a `SIGTTIN` (`SIGTTOU`) signal by the terminal driver, which, unless caught, causes the process to stop.

If the operating system on which `bash` is running supports job control, `bash` allows you to use it. Typing the suspend character (typically `^Z`, Control-Z) while a process is running causes that process to be stopped and returns you to `bash`. Typing the delayed suspend character (typically `^Y`, Control-Y) causes the process to be stopped when it attempts to read input from the terminal, and control to be returned to `bash`. You may then manipulate the state of this job, using the `bg` command to continue it in the background, the `fg` command to continue it in the foreground, or the `kill` command to kill it. A `^Z` takes effect immediately, and has the additional side effect of causing pending output and typeahead to be discarded.

There are a number of ways to refer to a job in the shell. The character `%` introduces a job name. Job number *n* may be referred to as `%n`. A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, `%ce` refers to a stopped `ce` job. If a prefix matches more than one job, `bash` reports an error. Using `%?ce`, on the other hand, would refer to any job containing the string `ce` in its command line. If the substring matches more than one job, `bash` reports an error. The symbols `%%` and `%+` refer to the shell's notion of the current job, which is the last job stopped while it was in the foreground. The previous job may be referenced using `%-`. In output pertaining to jobs (e.g. the output of the `jobs` command), the current job is always flagged with a `+`, and the previous job with a `-`.

Simply naming a job can be used to bring it into the foreground: `%1` is a synonym for `fg %1`, bringing job 1 from the background into the foreground. Similarly, `%1 &` resumes job 1 in the background, equivalent to `bg %1`.

The shell learns immediately whenever a job changes state. Normally, `bash` waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the variable `notify` is set, `bash` reports such changes immediately. (See also the `-o notify` option to the `set` builtin command.)

If you attempt to exit `bash` while jobs are stopped, the shell prints a message warning you. You may then use the `jobs` command to inspect their status. If you do this, or try to exit again immediately, you are not warned again, and the stopped jobs are terminated.

SIGNALS

When `bash` is interactive, it ignores `SIGTERM` (so that `kill 0` does not kill an interactive shell), and `SIGINT` is caught and handled (so that `wait` is interruptible). In all cases, `bash` ignores `SIGQUIT`. If job control is in effect, `bash` ignores `SIGTTIN`, `SIGTTOU`, and `SIGTSTP`.

Synchronous jobs started by `bash` have signals set to the values inherited by the shell from its parent. Background jobs (jobs started with `&`) ignore `SIGINT` and `SIGQUIT`. Commands run as a result of command substitution ignore the keyboard-generated job control signals `SIGTTIN`, `SIGTTOU`, and `SIGTSTP`.

COMMAND EXECUTION

After a command has been split into words, if it results in a simple command and an optional list of arguments, the following actions are taken.

If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described in "FUNCTIONS". If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.

If the name is neither a shell function nor a builtin, and contains no slashes, `bash` searches each element of the `PATH` for a directory containing an executable file by that name. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.

If the search is successful, or if the command name contains one or more slashes, the shell executes the named program. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments given, if any.

If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a shell script, a file containing shell commands. A subshell is spawned to execute it. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to handle the script, with the exception that the locations of commands remembered by the parent (see `hash` under "SHELL BUILTIN COMMANDS") are retained by the child.

If the program is a file beginning with `#!`, the remainder of the first line specifies an interpreter for the program. The shell executes the specified interpreter on operating systems that do not handle this executable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any.

ENVIRONMENT

When a program is invoked it is given an array of strings called the `environment`. This is a list of name-value pairs, of the form `name=value`.

The shell allows you to manipulate the environment in several ways. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for export to child processes. Executed commands inherit the environment. The `export` and `declare` **-x** commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the `unset` command, plus any additions via the `export` and `declare` **-x** commands.

The environment for any simple command or function may be augmented temporarily by prefixing it with parameter assignments, as described in "PARAMETERS". These assignment statements affect only the environment seen by that command.

If the **-k** flag is set, then all parameter assignments are placed in the environment for a command, not just those that precede the command name. See the `set` builtin command.

EXIT STATUS

For the purposes of the shell, a command which exits with a zero exit status has succeeded. An exit status of zero indicates success. A non-zero exit status indicates failure. When a command terminates on a fatal signal, `bash` uses the value of 128+`signal` as the exit status.

`bash` itself returns the exit status of the last command executed, unless a syntax error occurs, in which case it exits with a non-zero value. Also refer to the `exit` builtin command.

PROMPTING

When executing interactively, `bash` displays the primary prompt `PS1` when it is ready to read a command, and the secondary prompt `PS2` when it needs more input to complete a command. `bash` allows the prompt to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

`\t` the time

`\d`	the date
`\n`	CRLF
`\s`	the name of the shell, the basename of `$0` (the portion following the final slash)
`\w`	the current working directory
`\W`	the basename of the current working directory
`\u`	the username of the current user
`\h`	the hostname
`\#`	the command number of this command
`\!`	the history number of this command
`\$`	if the effective UID is 0, a `#`, otherwise a `$`
`\nnn`	character code in octal
`\\`	a backslash

After the string is decoded, if the variable `NO_PROMPT_VARS` is not set, it is expanded via parameter expansion, command substitution, arithmetic expansion, and word splitting.

READLINE

This is the library that handles reading input when using an interactive shell, unless the **`-nolineediting`** option is given. By default, the line editing commands are similar to those of Emacs. A *vi*-style line editing interface is also available.

In this section, the Emacs-style notation is used to denote keystrokes. **Control** keys are denoted by C-key (e.g.; `C-n` means **Control-n**. Similarly, Meta keys are denoted by `M-key`, so `M-x` means **Meta-x**. (On keyboards without a Meta key, `M-x` means **Esc x** (i.e., press the **Escape** key then the **x** key. The combination `M-C-x` means **Esc-Control-x**, or press the **Escape** key, then hold the **Control** key while pressing the **x** key.)

The default key-bindings may be changed with an `~/.inputrc` file. Other programs that use this library may add their own commands and bindings.

For example, placing

```
M-Control-u: universal_argument
```

or

```
C-Meta-u: universal_argument
```

into the

```
~/.inputrc
```

would make `M-C-u` execute the command `universal_argument`.

The following symbolic character names are recognized: `RUBOUT`, `DEL`, `ESC`, `NEWLINE`, `SPACE`, `RETURN`, `LFD`, `TAB`.

Placing

```
set editing-mode vi
```

into a

```
~/.inputrc
```

file causes `bash` to start with a *vi*-like editing mode. The editing mode may be switched during interactive use by using the **-o** option to the `set` builtin command.

You can have readline use a single line for display, scrolling the input between the two borders by placing

```
set horizontal-scroll-mode On
```

into a

```
~/.inputrc
```

file.

The following is a list of the names of the commands and the default keystrokes to get them.

Commands for Moving

`beginning-of-line (C-a)`
: Move to the start of the current line

`end-of-line (C-e)`
: Move to the end of the line

`forward-char (C-f)`
: Move forward a character

`backward-char (C-b)`
: Move back a character

`forward-word (M-f)`
: Move forward to the end of the next word

`backward-word (M-b)`
: Move back to the start of this, or the previous, word

`clear-screen (C-l)`
: Clear the screen leaving the current line at the top of the screen

Commands for Manipulating the History

`accept-line (Newline, Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the state of the `history_control` variable. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`
: Fetch the previous command from the history list, moving back in the list

`next-history (C-n)`
Fetch the next command from the history list, moving forward in the list

`beginning-of-history (M-<)`
Move to the first line in the history, the first line entered

`end-of-history (M->)`
Move to the end of the input history; i.e., the line you are entering

`reverse-search-history (C-r)`
Search backward starting at the current line and moving up through the history as necessary. This is an incremental search

`forward-search-history (C-s)`
Search forward starting at the current line and moving down through the history as necessary

`shell-expand-line (M-C-e)`
Expand the line the way the shell does when it reads it. This performs alias and history expansion. See "HISTORY" for more information.

`insert-last-argument (M-., M-_)`
Insert the last argument to the previous command (the last word on the previous line)

`operate-and-get-next (C-O)`
Accept the current line for execution and fetch the next line relative to the current line from the history file for editing

Commands for Changing Text

`delete-char (C-d)`
Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not `C-d`, then return `EOF`

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric `arg` says to kill the characters instead of deleting them

`quoted-insert (C-q, C-v)`
: Add the next character that you type to the line verbatim. This is how to insert characters like `C-q`, for example

`tab-insert (M-TAB)`
: Insert a tab character

`self-insert (a,b,A,1,!,...)`
: Insert the character typed

`transpose-chars (C-t)`
: Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work

`transpose-words (M-t)`
: Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well

`upcase-word (M-u)`
: Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point

`downcase-word (M-l)`
: Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move point

`capitalize-word (M-c)`
: Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move point

Killing and Yanking

`kill-line (C-k)`
: Kill the text from the current cursor position to the end of the line. This saves the killed text on the `kill-ring`. (see below)

`backward-kill-line`
: Kill backward to the beginning of the line. This is normally unbound, in favor of `unix-line-discard`, which emulates the behavior of the standard Unix terminal driver

`kill-word (M-d)`
: Kill from the cursor to the end of the current word, or if between words, to the end of the next word

`backward-kill-word (M-Rubout)`
: Kill the word behind the cursor

`unix-line-discard (C-u)`
: Do what C-u used to do in Unix line input. We save the killed text on the `kill-ring`, though

`unix-word-rubout (C-w)`
: Do what C-w used to do in Unix line input. The killed text is saved on the kill-ring. This is different than `backward-kill-word` because the word boundaries differ

`yank (C-y)`
: Yank the top of the kill ring into the buffer at point

`yank-pop (M-y)`
: Rotate the kill-ring, and yank the new top. Only works following yank or yank-pop

Arguments

`digit-argument (M-0, M-1, ..., M--)`
: Add this digit to the argument already accumulating, or start a new argument. `M--` starts a negative argument

`universal-argument`
Do what `C-u` does in *emacs.* By default, this is not bound to a key

Completing

`complete (TAB)`
Attempt to perform completion on the text before point. `bash` attempts completion treating the text as a variable (if the text begins with `$`), username (if the text begins with `~`), hostname (if the text begins with `@`), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.

`possible-completions (M-?)`
List the possible completions of the text before point

`complete-filename (M-/)`
Attempt filename completion on the text before point

`possible-filename-completions (C-x /)`
List the possible completions of the text before point, treating it as a filename

`complete-username (M-~)`
Attempt completion on the text before point, treating it as a username

`possible-username-completions (C-x ~)`
List the possible completions of the text before point, treating it as a username

`complete-variable (M-$)`
Attempt completion on the text before point, treating it as a shell variable

`possible-variable-completions (C-x $)`
List the possible completions of the text before point, treating it as a shell variable

`complete-hostname (M-@)`
Attempt completion on the text before point, treating it as a hostname

`possible-hostname-completions (C-x @)`
List the possible completions of the text before point, treating it as a hostname

Miscellaneous

`abort (C-g)` Abort the current editing command and ring the terminal's bell

`do-uppercase-version (M-a, M-b, ...)`
Run the command that is bound to the uppercased key

`prefix-meta (ESC)`
Metafy the next character typed. This is for people without a meta key. `ESC f` is equivalent to `Meta-f`

`undo (C-_)` Incremental undo, separately remembered for each line

`revert-line (M-r)`
Undo all changes made to this line. This is like typing the `undo` command enough times to get back to the beginning

`display-shell-version (C-x C-v)`
Display version information about the current instance of `bash`

`emacs-editing-mode (C-e)`
When in `vi` editing mode, this causes a switch to `emacs` editing mode

`vi-editing-mode (M-C-j or M-C-m)`
When in `emacs` editing mode, this causes a switch to `vi` editing mode

HISTORY

The shell supports a history expansion feature that is similar to the history expansion in `csh`. This section describes what syntax features are available.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is determining which line from the previous history to use during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is the *event*, and the portions of that line that are acted upon are *words*. The line is broken into words in the same fashion as when reading input, so that several English, or Unix, words surrounded by quotes are considered as one word. Only backslash (\) can quote the history escape character, which is `!` by default.

Event Designators

An event designator is a reference to a command line entry in the history list.

`!`	Start a history substitution, except when followed by a `<space>`, `<tab>`, `<newline>`, `=`, or `(`
`!!`	Refer to the previous command. This is a synonym for `!-1`
`!`*n*	Refer to command line *n*
`!-`*n*	Refer to the current command line minus *n*
`!`*string*	Refer to the most recent command starting with *string*
`!?`*string*`[?]`	Refer to the most recent command containing *string*

Word Designators

A `:` separates the event specification from the word designator. It can be omitted if the word designator begins with a `^`, `$`, `*`, or `%`. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`#`	The entire command line typed so far. This means the current command, not the previous command, so it really isn't a word designator, and does not belong in this section.
`0 (zero)`	The zeroth word. For the shell, this is the command word.
n	The *n*th word.

`^`	The first argument. That is, word 1.
`$`	The last argument.
`%`	The word matched by the most recent ?*string*? search.
x–*y*	A range of words; -*y* abbreviates 0-*y*.
`*`	All of the words but the zeroth. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event; the empty string is returned in that case.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:` (colon).

`h`	Remove a trailing pathname component, leaving only the head.
`r`	Remove a trailing suffix of the form `.xxx`, leaving the basename.
`e`	Remove all but the suffix.
`t`	Remove all leading pathname components, leaving the tail.
`p`	Print the new command but do not execute it. This takes effect immediately, so it should be the last specifier on the line.

ARITHMETIC EVALUATION

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the `let` built-in command and “Arithmetic Expansion”. Evaluation is done in long integers with no check for overflow, though division by 0 is trapped and flagged as an error. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

`-`	unary minus
`!`	logical NOT

`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`<= >= < >`	comparison
`== !=`	equality and inequality
`=`	assignment

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. The value of a parameter is coerced to a long integer within an expression. A shell variable need not have its integer attribute turned on to be used in an expression.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

SHELL BUILTIN COMMANDS

`:` [*arguments*]
No effect; the command does nothing beyond expanding *arguments* and performing any specified redirections. A zero exit code is returned.

`.` *filename*

`source` *filename*
Read and execute commands from *filename* in the current shell environment and return the exit status of the last command executed from *filename*. Pathnames in `PATH` are used to find the directory containing *filename*, if *filename* does not contain a slash. The file searched for in `PATH` need not be executable. The current directory is searched if no file is found in `PATH`. The return status is the status of the last command exited within the script (`true` if no commands are executed), and false if *filename* is not found.

`alias` [*name*[=*value*] ...]
`Alias` with no arguments prints the list of aliases in the form *name*=*value* on standard output. When arguments are supplied, an alias is defined for each *name* whose *value* is given. A trailing space in *value* causes the next

word to be checked for alias substitution when the alias is expanded. `Alias` returns true unless a *name* is given for which no alias has been defined.

`bg` [*jobspec*]
Place *jobspec* in the background, as if it had been started with `&`. If *jobspec* is not present, the shell's notion of the current job is used.

`bind` [`-lvd`] [`-q` *name*]

`bind -f` *filename*

`bind` *keyseq*:*function-name*
Display current `readline` key and function bindings, or bind a key sequence to a `readline` function or macro. The binding syntax accepted is identical to that of *.inputrc*, but each binding must be passed as a separate argument; (e.g., `"\C-x\C-r": re-read-init-file`.) Options, if supplied, have the following meanings:

-l List the names of all readline functions

-v List current function names and bindings

-d Dump function names and bindings in such a way that they can be re-read

-f *filename*
Read key bindings from *filename*

-q *function*
Query about which keys invoke the named *function*

`break` [*n*]
Exit from within a `for`, `while`, or `until` loop. If *n* is specified, break *n* levels. *n* must be `1`If *n* is greater than the number of enclosing loops, all enclosing loops are exited. The return value is `0` unless the shell is not executing a loop when `break` is executed.

`builtin` [*shell-builtin* [*arguments*]]
Execute the specified shell builtin, passing it *arguments*, and return its exit status. This is useful when you wish to define a function whose name is the same as a shell

builtin, but need the functionality of the builtin within the function itself. The `cd` builtin is commonly redefined this way.

`cd` [*dir*]

Change the current directory to *dir*. The variable `HOME` is the default *dir*. The variable `CDPATH` defines the search path for the directory containing *dir*. Alternative directory names are separated by a colon. A null directory name in `CDPATH` is the same as the current directory (i.e. **,** a period). If *dir* begins with a slash (`/`), then `CDPATH` is not used. An argument of – is equivalent to **$**`OLDPWD`. The return value is true if the directory was successfully changed; false otherwise.

`command` [**-p**] [*command* [*arg* ...]]

Run *command* with *args* suppressing the normal shell function lookup. Only builtin commands or commands found in the `PATH` are executed. If the **-p** option is given, the search for *command* is performed using a default value for `PATH` that is guaranteed to find all of the standard utilities. An argument of **--** disables option checking for the rest of the arguments. If an error occurred or *command* cannot be found, the exit status is 127. Otherwise, the exit status of the `command` builtin is the exit status of *command*.

`continue` [*n*]

Resume the next iteration of the enclosing `for`, `while`, or `until` loop. If *n* is specified, resume at the *n*th enclosing loop. *n* must be `1`. If *n* is greater than the number of enclosing loops, the last enclosing loop (the top-level loop) is resumed. The return value is `0` unless the shell is not executing a loop when `continue` is executed.

`declare` [**-frxi**] [*name*[=*value*]]

`typeset` [**-frxi**] [*name*[=*value*]]

Declare variables and/or give them attributes. If *name* is not given, display the values of variables instead.

-f Use function names only

-r Make *name* readonly. These names cannot then be assigned values by subsequent assignment statements.

-x Mark *name* for export to subsequent commands via the environment.

-i The variable is treated as an integer; arithmetic evaluation (See "ARITHMETIC EVALUATION" for more information.) is performed when the variable is assigned a value.

Using + instead of – turns off the attribute instead. When used in a function, makes *name*s local, as with the `local` command.

`dirs [-l]` Display the list of currently-remembered directories. Directories are added to the list with the `pushd` command; the `popd` command moves back up through the list. The **-l** option produces a longer listing; the default listing format uses a tilde to denote the home directory.

`echo` [**-ne**] [*arg* ...]

Output the *arg*s, separated by spaces. If **-n** is specified, the trailing newline is suppressed. If the **-e** option is given, interpretation of the following backslash-escaped characters is enabled:

\a	alert (bell)
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\nnn	the character whose ASCII code is *nnn* (octal)

`enable` **[-n]** [*name* ...]
: Enable and disable builtin shell commands. This allows the execution of a disk command which has the same name as a shell builtin without specifying a full pathname. If **-n** is used, each *name* is disabled; otherwise, *names* are enabled. For example, to use the `test` found in `PATH` instead of the shell builtin version, type `enable -n test`.

`eval` [*arg* ...]
: The *arg*s are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the `eval` command. If there are no *args*, or only null arguments, `eval` returns true.

`exec` [[-] *command* [*arguments*]]
: If *command* is specified, it replaces the shell. No new process is created. The *arguments* become the arguments to *command*. If the first argument is `-`, the shell places a dash in the zeroth arg passed to *command*. This is what login does. If the file cannot be executed for some reason, the shell exits, unless the shell variable `no_exit_on_failed_exec` exists. If *command* is not specified, any redirections take effect in the current shell.

`exit` [*n*]

`bye` [*n*]
: Cause the shell to exit with a status of *n*. If *n* is omitted, the exit status is that of the last command executed. A trap on `EXIT` is executed before the shell terminates.

`export` **[-npf]** [*name*[=*word*]] ...
: The supplied *names* are marked for automatic export to the environment of subsequently executed commands. If the **-f** option is given, the *names* refer to functions. If no *names* are given, or if the **-p** option is supplied, a list of all names that are exported in this shell is printed. The **-n** option causes the export property to be removed from the named variables. An argument of **--** disables option checking for the rest of the arguments. `export` returns an exit status of true unless an illegal option is encountered.

`fc` [**-e** *ename*] [**-nlr**] [*first*] [*last*]

`fc` **-s** [*pat*=*rep*] [*cmd*]

Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. *First* and *last* may be specified as a string (to locate the last command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified it is set to the current command for listing (so that `fc -l -10` prints the last 10 commands) and to *first* otherwise. If *first* is not specified it is set to the previous command for editing and -16 for listing.

The **-n** flag suppresses the command numbers when listing. The **-r** flag reverses the order of the commands. If the **-l** flag is given, the commands are listed on standard output. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the `FCEDIT` variable is used, and the value of `EDITOR` if `FCEDIT` is not set. If neither variable is set, `vi` is used. When editing is complete, the edited commands are echoed and executed.

In the second form, the command is re-executed after the substitution *old*=*new* is performed. A useful alias to use with this is `r=fc` **-s**, so that typing `r cc` runs the last command beginning with `cc` and typing `r` re-executes the last command.

`fg` [*jobspec*]

Place *jobspec* in the foreground, and make it the current job. If *jobspec* is not present, the shell's notion of the *current job* is used.

`getopts` *optstring name* [*args*]

`getopts` is used by shell procedures to parse positional parameters. *optstring* contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, `getopts` places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable `OPTIND`. `OPTIND` is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, `getopts` places that argument into the variable `OPTARG`. The shell does not reset `OPTIND` automatically; it must be manually reset between multiple calls to `getopts` within the same shell invocation if a new set of parameters is to be used.

`getopts` can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation diagnostic messages are printed when illegal options or missing option arguments are encountered. If the variable `OPTERR` is set to `0`, no error message will be displayed, even if the first character of *optstring* is not a colon.

If an illegal option is seen, `getopts` places `?` into *name* and, if not silent, prints an error message and unsets `OPTARG`. If `getopts` is silent, the option character found is placed in `OPTARG` and no diagnostic message is printed.

If a required argument is not found, and `getopts` is not silent, a question mark (`?`) is placed in *name*, `OPTARG` is unset, and a diagnostic message is printed. If `getopts` is silent, then a colon (`:`) is placed in *name* and `OPTARG` is set to the option character found.

`getopts` normally parses the positional parameters, but if more arguments are given in *args*, `getopts` parses those instead. `getopts` returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

`hash` [**-r**] [*name*]

For each *name*, the full pathname of the command is determined and remembered. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is printed. An argument of **--** disables option checking for the rest of the arguments. The return status is true unless a *name* is not found or an illegal option is supplied.

`help` [*pattern*] Display helpful information about builtin commands. If *pattern* is specified, `help` gives detailed help on all commands matching *pattern*; otherwise a list of the builtins is printed.

`history` [*n*]

`history` **-rwan** [*filename*]
With no options, display the command history list with line numbers. Lines listed with a `*` have been modified. An argument of *n* lists only the last *n* lines. If a non-option argument is supplied, it is used as the name of the history file; if not, the value of `HISTFILE` (default `~/.`*bash_history*) is used. Options, if supplied, have the following meanings:

-a Append the new history lines (history lines entered since the beginning of the current session) to the history file

-n Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current `bash` session.

-r Read the contents of the history file and use them as the current history

-w Write the current history to the history file, overwriting the history file's contents.

`jobs` [**-ln**p] [*jobspec* ...]

`jobs` **-x** *command* [*args* ...]
The first form lists the active jobs. The **-l** option lists process IDs in addition to the normal information; the **-p** option lists only the process ID of the job's process group leader. The **-n** option displays only jobs that have changed status since last notified. If *jobspec* is given, output is restricted to information about that job.

If the **-x** option is supplied, `jobs` replaces any *jobspec* found in *command* or *args* with the corresponding process group ID, and executes *command* passing it *args*.

`kill` [**-s** `sigspec` | **-sigspec**] [*pid* | *jobspec*] ...

`kill` **-l** [*signum*]

Send the signal named by *sigspec* to the processes named by *pid* or *jobspec*. *sigspec* is either a signal name such as `SIGKILL` or a signal number. If *sigspec* is a signal name, the name is case insensitive and may be given with or without the `SIG` prefix. If *sigspec* is not present, then `SIGTERM` is assumed. An argument of **-l** lists the signal names. If any arguments are supplied when **-l** is given, the names of the specified signals are listed. An argument of **--** disables option checking for the rest of the arguments. `kill` returns true if at least one signal was successfully sent, or false if an error occurs.

`let` *arg* [*arg* ...] Each *arg* is an arithmetic expression to be evaluated (See “ARITHMETIC EVALUATION” for more information.). If the last *arg* evaluates to 0, `let` returns `1`; `0` is returned otherwise.

`local` [*name*[=*value*]]

Create a local variable named *name*, and assign it *value*. When `local` is used within a function, it causes the variable *name* to have a visible scope restricted to that function and its children. With no operands, `local` writes a list of local variables to the standard output. It is an error to use `local` when not within a function.

`logout` Exit a login shell.

`popd` [+/-n] Removes entries from the directory stack. With no arguments, removes the top directory from the stack, and performs a `cd` to the new top directory.

`+n` Removes the *n*th entry counting from the left of the list shown by `dirs`, starting with zero. For example: `popd +0` removes the first directory, `popd +1` the second.

`-n` Removes the *n*th entry counting from the right of the list shown by `dirs`, starting with zero. For example: `popd -0` removes the last directory, `popd -1` the next to last.

If the variable `pushd_silent` is unset and the `popd` command is successful, a `dirs` is performed as well.

`pushd` *dir*

`pushd +/-n` Adds a directory to the top of the directory stack, or rotates the stack, making the new top of the stack the current working directory. With no arguments, exchanges the top two directories.

`+n` Rotates the stack so that the *n*th directory (counting from the left of the list shown by `dirs`) is at the top.

-n Rotates the stack so that the *n*th directory (counting from the right) is at the top.

`dir` Adds *dir* to the directory stack at the top, making it the new current working directory.

If the variable `pushd_silent` is not set and the `pushd` command is successful, a `dirs` is performed as well.

`pwd` Print the absolute pathname of the current working directory. The path printed contains no symbolic links (but see the description of `nolinks` under "Shell Variables").

`read` [**-r**] [*name* ...]
One line is read from the standard input, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. Only the characters in `IFS` are recognized as word delimiters. The return code is zero, unless end-of-file is encountered. If the **-r** option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line.

`readonly` [**-pf**] [*name* ...]
The given *names* are marked readonly and the values of these *names* may not be changed by subsequent assignment. If the **-f** option is supplied, the functions

corresponding to the *names* are so marked. If no arguments are given, or if the **-p** option is supplied, a list of all readonly names is printed. An argument of **--** disables option checking for the rest of the arguments.

`return` [*n*]
: Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script by the (source) command, it causes the shell to stop executing that script and return either *n* or the exit status of the last command executed within the script as the exit status of the script.

`set` [**-aefhknotuvxldCH**] [*arg* ...]
: The following list explains the effect of each of the hyphen-introduced options. These letter flags can also be specified as command-line options when running a new shell. The currently active options may be found in the variable `$-`.

Using + rather than – to introduce an option deactivates the option.

After the option arguments are processed, the remaining *arg*s are treated as values for the positional parameters and are assigned, in order, to `$1, $2, ...$9`. If no options or *arg*s are supplied, all shell variables are printed. The return status is always true unless an illegal option is encountered.

-a Automatically mark variables which are modified or created for export to the environment of subsequent commands.

-e Exit immediately if a simple-command (See "SHELL GRAMMAR" for more information.) exits with a non-zero status. The shell does not exit if the command that fails is part of an until or while loop, part of an if statement, part of a && or || list, or if the command's return value is being inverted via !.

-f Disable pathname expansion.

-h Locate and remember function commands as functions are defined. Function commands are normally looked up when the function is executed.

-k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

-m Monitor mode. Job control is enabled. This flag is on by default for interactive shells on systems that support it. (See “JOB CONTROL” for more information.) Background processes run in a separate process group and a line containing their exit status is printed upon their completion.

-n Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored for interactive shells.

-o *option_name*
If no *option-name* is supplied, the values of the current options are printed. The *option_name* can be one of the following:

`allexport`	Same as **-a**
`braceexpand`	The shell performs curly brace expansion. (See “Brace Expansion” for more information.) This is on by default.
`emacs`	Use an Emacs-style command line editing interface.
`errexit`	Same as **-e**
`histexpand`	Same as **-H**
`ignoreeof`	The effect is as if the shell command `IGNOREEOF=10` had been executed. (See “Shell Variables” for more information.)
`monitor`	Same as **-m**

noclobber	Same as **-C**
noexec	Same as **-n**
noglob	Same as **-f**
nohash	Same as **-d**
notify	The effect is as if the shell command notify= had been executed. (See "Shell Variables" for more information.)
nounset	Same as **-u**
verbose	Same as **-v**
vi	Use a vi-style command line editing interface.
xtrace	Same as **-x**

-t Exit after reading and executing one command.

-u Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, the shell prints an error message, and, if not interactive, exits with a non-zero status.

-v Print shell input lines as they are read.

-x After expanding each simple-command, bash displays the expanded value of PS4, followed by the command and its expanded arguments.

-l Save and restore the binding of *name* in a for *name* [in word] command. (See "SHELL GRAMMAR" for more information.)

-d Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.

-C	The effect is as if the shell command `noclobber=` had been executed. (See "Shell Variables" for more information.)
-H	Enable ! style history substitution. This flag is on by default.
--	If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the `args`, even if some of them begin with a `-`.
-	Signal the end of options, cause all remaining `args` to be assigned to the positional parameters. The **-x** and **-v** options are turned off. If there are no `args`, the positional parameters remain unchanged.

`shift` [*n*]
: The positional parameters from *n*`+1 ...` are renamed to `$1 ....` If *n* is not given, it is assumed to be `1`. The exit status is `1` if *n* is greater than `$#`; otherwise 0.

`suspend` [**-f**]
: Suspend the execution of this shell until it receives a `SIGCONT` signal. The **-f** option says not to complain if this is a login shell; just suspend anyway.

`test` *expr*

`[`*expr*`]`
: Return a status of `0` (true) or `1` (false) depending on the evaluation of the conditional expression *expr*. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well.

-b *file*	True if *file* exists and is block special.
-c *file*	True if *file* exists and is character special.
-d *file*	True if *file* exists and is a directory.
-e *file*	True if *file* exists
-f *file*	True if *file* exists and is a regular file.
-g *file*	True if *file* exists and is set_group_id.

-k *file* True if *file* has its sticky bit set.

-L *file* True if *file* exists and is a symbolic link.

-p *file* True if *file* exists and is a named pipe.

-r *file* True if *file* exists and is readable.

-s *file* True if *file* exists and has a size greater than zero.

-S *file* True if *file* exists and is a socket.

-t [*fd*] True if *fd* is opened on a terminal. If *fd* is omitted, it defaults to `1` (standard output).

-u *file* True if *file* exists and its set-user-id bit is set.

-w *file* True if *file* exists and is writable.

-x *file* True if *file* exists and is executable.

-O *file* True if *file* exists and is owned by the effective user id.

-G *file* True if *file* exists and is owned by the effective group id.

file1 **-nt** *file2*
True if *file1* is newer (according to modification date) than *file2*.

file1 **-ot** *file2*
True if *file1* is older than *file2*.

file1 **-ef** *file*
True if *file1* and *file2* have the same device and inode numbers.

-z *string*
True if the length of *string* is zero.

-n *string* True if the length of *string* is non-zero.

string1 = *string2*
True if the strings are equal.

string1 != *string2*
True if the strings are not equal.

! *expr* True if *expr* is false.

expr1 **-a** *expr2*
True if both *expr1* AND *expr2* are true.

expr1 **-o** *expr2*
True if either *expr1* OR *expr2* is true.

arg1 OP *arg2*
OP is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if *arg1* is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than *arg2*, respectively. *arg1* and *arg2* may be positive integers, negative integers, or the special expression **-l** *string*, which evaluates to the length of *string*.

`times` Print the accumulated user and system times for the shell and for processes run from the shell.

`trap` [*arg*] [*sigspec*]
The command *arg* is to be read and executed when the shell receives signal(s) *sigspec*. If *arg* is absent or -, all specified signals are reset to their original values (the values they had upon entrance to the shell). If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. *sigspec* is either a signal name in `<signal.h>`, or a signal number. If *sigspec* is `EXIT` (0) the command *arg* is executed on exit from the shell.

With no arguments, `trap` prints the list of commands associated with each signal number. The **-l** option causes the shell to print a list of signal names and their corresponding numbers. An argument of **--** disables option checking for the rest of the arguments. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if either then trap name or number is invalid; otherwise `trap` returns true.

`type` [**-all**] [**-type** | **-path**] [*name* ...]
With no options, indicate how each *name* would be interpreted if used as a command name. If the **-type** flag is used, `type` prints a phrase which is one of `alias`,

`keyword`, `function`, `builtin`, or `file` if *name* is an alias, shell reserved word, function, builtin, or disk file, respectively. If the name is not found, then nothing is printed, and an exit status of false is returned. If the **-path** flag is used, `type` either returns the name of the disk file that would be executed if *name* were specified as a command name, or nothing if **-type** would not return *file*. If a command is hashed, **-path** prints the hashed value, not necessarily the file that appears first in `PATH`. If the **-all** flag is used, `type` prints all of the places that contain an executable named *name*.

This includes aliases and functions, if and only if the **-path** flag is not also used. The table of hashed commands is not consulted when using **-all**. **type** accepts **-a**, **-t**, and **-p** in place of **-all**, **-type**, and **-path**, respectively. An argument of **--** disables option checking for the rest of the arguments. `type` returns true if any of the arguments are found, false if none are found.

`ulimit` [**-SHacdfmstpn** [*limit*]]

`Ulimit` provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The value of *limit* can be a number in the unit specified for the resource, or the value `unlimited`. The `H` and `S` options specify that the hard or soft limit is set for the given resource.

A hard limit cannot be increased once it is set; a soft limit may be increased up to the value of the hard limit. If neither `H` nor `S` is specified, the command applies to the soft limit. If *limit* is omitted, the current value of the soft limit of the resource is printed, unless the `H` option is given. When more than one resource is specified, the limit name and unit is printed before the value. Other options are interpreted as follows:

-a All current limits are reported

-c The maximum size of core files created

-d The maximum size of a process's data segment

-f	The maximum size of files created by the shell
-m	The maximum resident set size
-s	The maximum stack size
-t	The maximum amount of cpu time in seconds
-p	The pipe size in 512-byte blocks (this may not be set)
-n	The maximum number of open file descriptors (most systems do not allow this value to be set, only displayed)

An argument of **--** disables option checking for the rest of the arguments. If *limit* is given, it is the new value of the specified resource (the **-a** option is display only). If no option is given, then **-f** is assumed. Values are in 1024-byte increments, except for **-t**, which is in seconds, and **-p**, which is in units of 512-byte blocks.

`umask` [**-S**] [*mode*]

The user file-creation mask is set to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by `chmod`. If *mode* is omitted, or if the **-S** option is supplied, the current value of the mask is printed. The **-S** option causes the mask to be printed in symbolic form; the default output is an octal number. An argument of **--** disables option checking for the rest of the arguments.

`unalias` [*name* ...]

Remove *name*s from the list of defined aliases. The return value is true unless *name* is not a defined alias.

`unset` [**-fv**] [*name* ...]

For each *name*, remove the corresponding variable or, given the **-f** option, function. An argument of **--** disables option checking for the rest of the arguments. Note that `PATH`, `IFS`, `PPID`, `PS1`, `PS2`, `UID`, and `EUID` cannot be unset. If any of `RANDOM`, `SECONDS`, or `LINENO` are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless the variable *name* does not exist or is non-unsettable.

`wait` [*n*] Wait for the specified process and report its termination status. *n* may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If *n* is not given, all currently active child processes are waited for, and the return code is zero.

INVOCATION

A *login shell* is one whose first character of argument zero is a `-`, or one started with the **`-login`** flag.

An *interactive* shell is one whose standard input and output are both connected to terminals (as determined by `isatty()`), or one started with the **`-i`** flag. `PS1` is set and `$-` includes `i` if `bash` is interactive, allowing a way to test this state from a shell script or a startup file.

Login shells:
On login:
    if `/etc/profile` exists, source it.

        if `~/.`*bash_profile* exists, source it,
            else if `~/.`*bash_login* exists, source it,
                else if `~/.profile` exists, source it.

  On logout:
    if `~/.`*bash_logout* exists, source it

Non-login interactive shells:
  On startup:
    if `~/.bashrc` exists, source it.

Non-interactive shells:
  On startup:
    if the environment variable ENV is non-null, expand it and source the file it names.

FILES

`/bin/bash`	The bash executable
`/etc/profile`	The system-wide initialization file, executed for login shells
`~/.bash_profile`	The personal initialization file, executed for login shells
`~/.bashrc`	The individual per-interactive-shell startup file
`~/.inputrc`	Individual *Readline* initialization file

SEE ALSO

Utility Programs - `sh, ksh, csh`

AUTHORS

Brian Fox, Free Software Foundation (primary author)
`bfox@ai.MIT.Edu`

Chet Ramey, Case Western Reserve University
`chet@ins.CWRU.Edu`

BUG REPORTS

If you find a bug in `bash`, you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of `bash` that you have.

Once you have determined that a bug actually exists, mail a bug report to bash-maintainers@ai.MIT.Edu. If you have a fix, you are welcome to mail that as well! Suggestions and 'philosophical' bug reports may be mailed to bug-bash@ai.MIT.Edu or posted to the Usenet newsgroup gnu.bash.bug.

ALL bug reports should include:

- The version number of `bash`
- The hardware and operating system
- The compiler used to compile
- A description of the bug behavior
- A short script or recipe which exercises the bug
- Comments and bug reports concerning this manual page should be directed to `chet@ins.CWRU.Edu`

BUGS

It's too big and too slow.

There are some subtle differences between bash and traditional versions of sh, mostly because of the POSIX specification.

Aliases are confusing in some uses.

COPYRIGHT

NAME

`bc` – an arbitrary precision calculator language

SYNOPSIS

`bc` [**`-lws`**] [*file* ...]

VERSION

This man page documents GNU `bc` version 1.02.

DESCRIPTION

`bc` is a language that supports arbitrary precision numbers with interactive execution of statements. There are some similarities in the syntax to the C programming language. A standard math library is available by command line option. If requested, the math library is defined before processing any files. `bc` starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, `bc` reads from the standard input. All code is executed as it is read. (If a file contains a command to halt the processor, `bc` will never read from the standard input.)

This version of `bc` contains several extensions beyond traditional `bc` implementations and the POSIX draft standard. Command line options can cause these extensions to print a warning or to be rejected. This document describes the language accepted by this processor. Extensions will be identified as such.

OPTIONS

`-l` Define the standard math library.

`-w` Give warnings for extensions to POSIX `bc`.

`-s` Process exactly the POSIX `bc` language.

NUMBERS

The most basic element in `bc` is the number. Numbers are arbitrary precision numbers. This precision is both in the integer part and the fractional part. All numbers are represented internally in decimal and all computation is done in decimal. (This version truncates results from divide and multiply operations.) There are two attributes of numbers, the length and the scale. The length is the total number of significant decimal digits in a number and the scale is the total number of decimal digits after the decimal point. For example:

.000001 has a length of 6 and scale of 6.
1935.000 has a length of 7 and a scale of 3.

VARIABLES

Numbers are stored in two types of variables, simple variables and arrays. Both simple variables and array variables are named. Names begin with a letter followed by any number of letters, digits and underscores. All letters must be lower case. (Full alpha-numeric names are an extension. In POSIX `bc` all names are a single lower case letter.) The type of variable is clear by the context because all array variable names will be followed by brackets (`[]`).

There are four special variables, `scale`, `ibase`, `obase`, and `last`. `scale` defines how some operations use digits after the decimal point. The default value of `scale` is `0`. `ibase` and `obase` define the conversion base for input and output numbers. The default for both input and output is base `10`. `last` (an extension) is a variable that has the value of the last printed number. These will be discussed in further detail where appropriate. All of these variables may have values assigned to them as well as used in expressions.

COMMENTS

Comments in `bc` start with the characters `/*` and end with the characters `*/`. Comments may start anywhere and appear as a single space in the input. (This causes comments to delimit other input items. For example, a comment can not be found in the middle of a variable name.) Comments include any newlines (end of line) between the start and the end of the comment.

EXPRESSIONS

The numbers are manipulated by expressions and statements. Since the language was designed to be interactive, statements and expressions are executed as soon as possible. There is no main program. Instead, code is executed as it is encountered. (Functions, discussed in detail later, are defined when encountered.)

A simple expression is just a constant. `bc` converts constants into internal decimal numbers using the current input base, specified by the variable `ibase`. (There is an exception in functions.) The legal values of `ibase` are `2` through `16 (F)`. Assigning a value outside this range to `ibase` will result in a value of `2` or `16`. Input numbers may contain the characters 0-`9` and `A–F`. (Note: They must be capitals. Lower case letters are variable names.) Single digit numbers always have the value of the digit regardless of the value of `ibase`. (i.e. `A = 10`.) For multi-digit numbers, `bc` changes all input digits greater or equal to ibase to the value of `ibase-1`. This makes the number `FFF` always be the largest 3-digit number of the input base.

Full expressions are similar to many other high level languages. Since there is only one kind of number, there are no rules for mixing types. Instead, there are rules on the scale of expressions. Every expression has a scale. This is derived from the scale of original numbers, the operation performed and in many cases, the value of the variable `scale`. Legal values of the variable `scale` are `0` to the maximum number representable by a C integer.

In the following descriptions of legal expressions, `expr` refers to a complete expression and `var` refers to a simple or an array variable. A simple variable is just a

name

and an array variable is specified as

name[*expr*]

Unless specifically mentioned the scale of the result is the maximum scale of the expressions involved.

`- expr`	The result is the negation of the expression.

`++ var`	The variable is incremented by one and the new value is the result of the expression.
`-- var`	The variable is decremented by one and the new value is the result of the expression.
`var ++`	The result of the expression is the value of the variable and then the variable is incremented by one.
`var --`	The result of the expression is the value of the variable and then the variable is decremented by one.
`expr + expr`	The result of the expression is the sum of the two expressions.
`expr - expr`	The result of the expression is the difference of the two expressions.
`expr * expr`	The result of the expression is the product of the two expressions.
`expr / expr`	The result of the expression is the quotient of the two expressions. The scale of the result is the value of the variable `scale`.
`expr % expr`	The result of the expression is the remainder and it is computed in the following way. To compute `a%b`, first `a/b` is computed to `scale` digits. That result is used to compute `a-(a/b)*b` to the scale of the maximum of `scale+scale(b)` and `scale(a)`. If `scale` is set to zero and both expressions are integers this expression is the integer remainder function.
`expr ^ expr`	The result of the expression is the value of the first raised to the second. The second expression must be an integer. (If the second expression is not an integer, a warning is generated and the expression is truncated to get an integer value.) The scale of the result is `scale` if the exponent is negative. If the exponent is positive the scale of the result is the minimum of the scale of the first expression times the value of the exponent and the maximum of `scale` and the scale of the first expression. (e.g. `scale(a^b) = min(scale(a)*b, max(scale, scale(a)))`.) It should be noted that `expr^0` will always return the value of `1`.

`(expr)` This alters the standard precedence to force the evaluation of the expression.

`var = expr` The variable is assigned the value of the expression.

`var <op>= expr`
This is equivalent to `var = var <op> expr` with the exception that the `var` part is evaluated only once. This can make a difference if `var` is an array.

Relational expressions are a special kind of expression that always evaluate to 0 or 1, 0 if the relation is false and 1 if the relation is true. These may appear in any legal expression. (POSIX bc requires that relational expressions are used only in if, while, and for statements and that only one relational test may be done in them.) The relational operators are

`expr1 < expr2` The result is 1 if `expr1` is strictly less than `expr2`.

`expr1 <= expr2`
The result is 1 if `expr1` is less than or equal to `expr2`.

`expr1 > expr2` The result is 1 if `expr1` is strictly greater than `expr2`.

`expr1 >= expr2`
The result is 1 if `expr1` is greater than or equal to `expr2`.

`expr1 == expr2`
The result is 1 if `expr1` is equal to `expr2`.

`expr1 != expr2`
The result is 1 if `expr1` is not equal to `expr2`.

Boolean operations are also legal. (POSIX `bc` does NOT have boolean operations). The result of all boolean operations are `0` and `1` (for false and true) as in relational expressions. The boolean operators are:

`!expr` The result is `1` if *expr* is `0`.

`expr && expr` The result is `1` if both expressions are non-zero.

`expr || expr` The result is `1` if either expression is non-zero.

The expression precedence is as follows: (lowest to highest)

|| operator, left associative
&& operator, left associative
! operator, non-associative
Relational operators, left associative
Assignment operator, right associative
+ and - operators, left associative
*, / and % operators, left associative
^ operator, right associative
unary - operator, non-associative
++ and -- operators, non-associative

This precedence was chosen so that POSIX compliant `bc` programs will run correctly. This will cause the use of the relational and logical operators to have some unusual behavior when used with assignment expressions. Consider the expression:

```
a = 3 < 5
```

Most C programmers would assume this would assign the result of `3 < 5` (the value 1) to the variable `a`. What this does in `bc` is assign the value 3 to the variable `a` and then compare 3 to 5. It is best to use parenthesis when using relational and logical operators with the assignment operators.

There are a few more special expressions that are provided in `bc`. These have to do with user defined functions and standard functions. They all appear as *name*(*parameters*). See the section on functions for user defined functions. The standard functions are:

length (*expression*)
: The value of the length function is the number of significant digits in the expression.

read ( )
: The read function (an extension) will read a number from the standard input, regardless of where the function occurs. Beware, this can cause problems with the mixing of data and program in the standard input. The best use for this function is in a previously written program that needs input from the user, but never allows program code to be input from the user. The value of the read function is the number read from the standard input using the current value of the variable `ibase` for the conversion base.

scale (*expression*)
: The value of the scale function is the number of digits after the decimal point in the expression.

`sqrt` (*expression*)
: The value of the `sqrt` function is the square root of the expression. If the expression is negative, a run time error is generated.

STATEMENTS

Statements (as in most algebraic languages) provide the sequencing of expression evaluation. In `bc` statements are executed as soon as possible. Execution happens when a newline in encountered and there is one or more complete statements. Due to this immediate execution, newlines are very important in `bc`. In fact, both a semicolon and a newline are used as statement separators. An improperly placed newline will cause a syntax error. Because newlines are statement separators, it is possible to hide a newline by using the backslash character. The sequence `\<nl>`, where `<nl>` is the newline appears to `bc` as whitespace instead of a newline. A statement list is a series of statements separated by semicolons and newlines. The following is a list of `bc` statements and what they do: (Things enclosed in brackets ([]) are optional parts of the statement.)

expression
: This statement does one of two things. If the expression starts with `<variable> <assignment>...`, it is considered to be an assignment statement. If the expression is not an assignment statement, the expression is evaluated and printed to the output.

: After the number is printed, a newline is printed. For example, `a=1` is an assignment statement and `(a=1)` is an expression that has an embedded assignment. All numbers that are printed are printed in the base specified by the variable `obase`. The legal values for `obase` are 2 through `BC_BASE_MAX`. (See the section LIMITS.) For bases 2 through 16, the usual method of writing numbers is used. For bases greater than 16, `bc` uses a multi-character digit method of printing the numbers where each higher base digit is printed as a base 10 number.

The multi-character digits are separated by spaces. Each digit contains the number of characters required to represent the base ten value of `obase-1`. Since numbers are of arbitrary precision, some numbers may not be printable on a single output line. These long numbers will be split across lines using the \ as the last character on a line. The maximum number of characters printed per line is 70.

Due to the interactive nature of `bc` printing a number cause the side effect of assigning the printed value the special variable `last`. This allows the user to recover the last value printed without having to retype the expression that printed the number. Assigning to `last` is legal and will overwrite the last printed value with the assigned value. The newly assigned value will remain until the next number is printed or another value is assigned to `last`.

string — The `string` is printed to the output. Strings start with a double quote character and contain all characters until the next double quote character. All characters are take literally, including any newline. No newline character is printed after the string.

`print` *list* — The `print` statement (an extension) provides another method of output. The `list` is a list of strings and expressions separated by commas. Each string or expression is printed in the order of the list. No terminating newline is printed. Expressions are evaluated and their value is printed and assigned the variable `last`. Strings in the print statement are printed to the output and may contain special characters. Special characters start with the backslash character (\). The special characters recognized by `bc` are `b` (bell), `f` (form feed), `n` (newline), `r` (carriage return), `t` (tab), and `\` (backslash). Any other character following the backslash will be ignored. This still does not allow the double quote character to be part of any string.

{*statement_list*} — This is the compound statement. It allows multiple statements to be grouped together for execution.

`if` (*expression*) *then statement1* [*else statement2*]
: The `if` statement evaluates the expression and executes *statement1* or *statement2* depending on the value of the expression. If the expression is non-zero, *statement1* is executed. If *statement2* is present and the value of the expression is 0, then *statement2* is executed. (The else clause is an extension.)

`while` (*expression*) *statement*
: The `while` statement will execute the statement while the expression is non-zero. It evaluates the expression before each execution of the statement. Termination of the loop is caused by a zero expression value or the execution of a break statement.

`for` ([*expression1*] ; [*expression2*] ; [*expression3*]) *statement*
: The `for` statement controls repeated execution of the statement. *expression1* is evaluated before the loop. *expression2* is evaluated before each execution of the statement. If it is non-zero, the statement is evaluated. If it is zero, the loop is terminated. After each execution of the statement, *expression3* is evaluated before the reevaluation of *expression2*. If *expression1* or *expression3* are missing, nothing is evaluated at the point they would be evaluated. If *expression2* is missing, it is the same as substituting the value 1 for *expression2*. (The optional expressions are an extension. POSIX `bc` requires all three expressions.) The following is equivalent code for the for statement:

```
expression1;
while (expression2) {
   statement;
   expression3;
}
```

`break`
: This statement causes a forced exit of the most recent enclosing while statement or for statement.

`continue`
: The `continue` statement (an extension) causes the most recent enclosing for statement to start the next iteration.

`halt` The `halt` statement (an extension) is an executed statement that causes the `bc` processor to quit only when it is executed. For example, `if (0 == 1) halt` will not cause `bc` to terminate because the halt is not executed.

`return` Return the value 0 from a function. (See the section on functions.)

`return` (*expression*)
Return the value of the expression from a function. (See the section on functions.)

PSEUDO STATEMENTS

These statements are not statements in the traditional sense. They are not executed statements. Their function is performed at compile time.

`limits` Print the local limits enforced by the local version of `bc`. This is an extension.

`quit` When the quit statement is read, the `bc` processor is terminated, regardless of where the quit statement is found. For example, `if (0 == 1)` quit will cause `bc` to terminate.

`warranty` Print a longer warranty notice. This is an extension.

FUNCTIONS

Functions provide a method of defining a computation that can be executed later. Functions in `bc` always compute a value and return it to the caller. Function definitions are dynamic in the sense that a function is undefined until a definition is encountered in the input. That definition is then used until another definition function for the same name is encountered. The new definition then replaces the older definition. A function is defined as follows:

define *name (parameters) {newline*
auto_list statement_list}

A function call is just an expression of the form *name*(*parameters*).

Parameters are numbers or arrays (an extension). In the function definition, zero or more parameters are defined by listing their names separated by commas. Numbers are only call by value parameters. Arrays are only call by variable. Arrays are specified in the parameter definition by the notation *name*[]. In the function call, actual parameters are full expressions for number parameters.

The same notation is used for passing arrays as for defining array parameters. The named array is passed by variable to the function. Since function definitions are dynamic, parameter numbers and types are checked when a function is called. Any mismatch in number or types of parameters will cause a runtime error. A runtime error will also occur for the call to an undefined function.

The *auto_list* is an optional list of variables that are for local use. The syntax of the auto list (if present) is auto name, ... ;. (The semicolon is optional.) Each *name* is the name of an auto variable. Arrays may be specified by using the same notation as used in parameters. These variables have their values pushed onto a stack at the start of the function. The variables are then initialized to zero and used throughout the execution of the function.

At function exit, these variables are popped so that the original value (at the time of the function call) of these variables are restored. The parameters are really auto variables that are initialized to a value provided in the function call. Auto variables are different than traditional local variables in the fact that if function A calls function B, B may access function A's auto variables by just using the same name, unless function B has called them auto variables. Due to the fact that auto variables and parameters are pushed onto a stack, `bc` supports recursive functions.

The function body is a list of `bc` statements. Again, statements are separated by semicolons or newlines. Return statements cause the termination of a function and the return of a value. There are two versions of the return statement. The first form, `return`, returns the value 0 to the calling expression. The second form, `return (`*expression*`)`, computes the value of the expression and returns that value to the calling expression. There is an implied `return (0)` at the end of every function. This allows a function to terminate and return 0 without an explicit return statement.

Functions also change the usage of the variable `ibase`. All constants in the function body will be converted using the value of `ibase` at the time of the function call. Changes of `ibase` will be ignored during the execution of the function except for the standard function `read`, which will always use the current value of `ibase` for conversion of numbers.

MATH LIBRARY

If `bc` is invoked with the `-l` option, a math library is preloaded and the default scale is set to `20`. The math functions will calculate their results to the scale set at the time of their call. The math library defines the following functions:

s (*x*)	The sine of *x* in radians.
c (*x*)	The cosine of *x* in radians.
a (*x*)	The arctangent of *x*.
l (*x*)	The natural logarithm of *x*.
e (*x*)	The exponential function of raising `e` to the value *x*.
j (*n*,*x*)	The bessel function of integer order *n* of *x*.

EXAMPLES

In `/bin/sh`, the following will assign the value of `pi` to the shell variable `pi`.

```
pi=$(echo "scale=10; 4*a(1)" | bc -l)
```

The following is the definition of the exponential function used in the math library. This function is written in POSIX bc.

```
scale = 20
/* Uses the fact that e^x = (e^(x/2))^2
   When x is small enough, we use the series:
     e^x = 1 + x + x^2/2! + x^3/3! + ...
*/
define e(x) {
  auto  a, d, e, f, i, m, v, z
  /* Check the sign of x. */
  if (x<0) {
    m = 1
    x = -x
  }
  /* Precondition x. */
  z = scale;
  scale = 4 + z + .44*x;
  while (x > 1) {
    f += 1;
    x /= 2;
  }
  /* Initialize the variables. */
  v = 1+x
  a = x
  d = 1
  for (i=2; 1; i++) {
    e = (a *= x) / (d *= i)
    if (e == 0) {
      if (f>0) while (f--)  v = v*v;
      scale = z
      if (m) return (1/v);
      return (v/1);
    }
    v += e
  }
}
```

The following is code that uses the extended features of `bc` to implement a simple program for calculating checkbook balances. This program is best kept in a file so that it can be used many times without having to retype it at every use.

```
scale=2
print "\nCheck book program!\n"
print "  Remember, deposits are negative transactions.\n"
print "  Exit by a 0 transaction.\n\n"
print "Initial balance? "; bal = read()
bal /= 1
print "\n"
while (1) {
  "current balance = "; bal
  "transaction? "; trans = read()
  if (trans == 0) break;
  bal -= trans
  bal /= 1
}
quit
```

The following is the definition of the recursive factorial function.

```
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
}
```

DIFFERENCES

This version of `bc` was implemented from the POSIX P1003.2/D11 draft and contains several differences and extensions relative to the draft and traditional implementations. It is not implemented in the traditional way using `dc`. This version is a single process which parses and runs a byte code translation of the program. There is an undocumented option (`-c`) that causes the program to output the byte code to the standard output instead of running it. It was mainly used for debugging the parser and preparing the math library.

A major source of differences is extensions, where a feature is extended to add more functionality and additions, where new features are added. The following is the list of differences and extensions. `.IP LANG environment`.

This version does not conform to the POSIX standard in the processing of the `LANG` environment variable and all environment variables starting with `LC_`.

`names` — Traditional and POSIX `bc` have single letter names for functions, variables and arrays. They have been extended to be multi-character names that start with a letter and may contain letters, numbers and the underscore character.

`Strings` — Strings are not allowed to contain NUL characters. POSIX says all characters must be included in strings.

`last` — POSIX `bc` does not have a `last` variable. Some implementations of `bc` use the period (.) in a similar way.

`comparisons` — POSIX `bc` allows comparisons only in the if statement, the while statement, and the second expression of the for statement. Also, only one relational operation is allowed in each of those statements.

`if statement, else clause` — POSIX `bc` does not have an else clause.

`for statement` — POSIX `bc` requires all expressions to be present in the for statement.

`&&, ||, !` — POSIX `bc` does not have the logical operators.

`read function` — POSIX `bc` does not have a read function.

`print statement` — POSIX `bc` does not have a print statement.

`continue statement` — POSIX `bc` does not have a continue statement.

`array parameters` — POSIX `bc` does not have array parameters. Other implementations of `bc` may have call by value array parameters.

`=+, =-, =*, =/, =%, =^`
: POSIX `bc` does not require these old style assignment operators to be defined. This version may allow these old style assignments. Use the limits statement to see if the installed version supports them. If it does support the old style assignment operators, the statement `a =- 1` will decrement `a` by `1` instead of setting `a` to the value `-1`.

`spaces in numbers`
: Other implementations of `bc` allow spaces in numbers. For example, `x=1 3` would assign the value `13` to the variable *x*. The same statement would cause a syntax error in this version of `bc`.

`errors and execution`
: This implementation varies from other implementations in terms of what code will be executed when syntax and other errors are found in the program. If a syntax error is found in a function definition, error recovery tries to find the beginning of a statement and continue to parse the function. Once a syntax error is found in the function, the function will not be callable and becomes undefined. Syntax errors in the interactive execution code will invalidate the current execution block. The execution block is terminated by an end of line that appears after a complete sequence of statements. For example,

```
a = 1
b = 2
```

has two execution blocks and

```
{ a = 1
b = 2 }
```

has one execution block. Any runtime error will terminate the execution of the current execution block. A runtime warning will not terminate the current execution block.

`Interrupts`
: During an interactive session, the `SIGINT` signal (usually generated by the `control-C` character from the terminal) will cause execution of the current execution

block to be interrupted. It will display a runtime error indicating which function was interrupted. After all runtime structures have been cleaned up, a message will be printed to notify the user that `bc` is ready for more input. All previously defined functions remain defined and the value of all non-auto variables are the value at the point of interruption. All auto variables and function parameters are removed during the clean-up process. During a non-interactive session, the `SIGINT` signal will terminate the entire run of `bc`.

LIMITS

The following are the limits currently in place for this `bc` processor. Some of them may have been changed by an installation. Use the limits statement to see the actual values.

`BC_BASE_MAX` The maximum output base is currently set at 999. The maximum input base is `16`.

`BC_DIM_MAX` This is currently an arbitrary limit of `65535` as distributed. Your installation may be different.

`BC_SCALE_MAX` The number of digits after the decimal point is limited to `INT_MAX` digits. Also, the number of digits before the decimal point is limited to `INT_MAX` digits.

`BC_STRING_MAX` The limit on the number of characters in a string is `INT_MAX` characters.

`exponent` The value of the exponent in the raise operation (`^`) is limited to `LONG_MAX`.

`multiply` The multiply routine may yield incorrect results if a number has more than `LONG_MAX / 90` total digits. For 32-bit longs, this number is `23,860,929` digits.

`code size` Each function and the main program are limited to 10240 bytes of compiled byte code each. This limit (`BC_MAX_SEGS`) can be easily changed to have more than 10 segments of 1024 bytes.

`variable names` The current limit on the number of unique names is `32767` for each of simple variables, arrays and functions.

FILES

In most installations, `bc` is completely self-contained. Where executable size is of importance or the C compiler does not deal with very long strings, `bc` will read the standard math library from the file `/usr/local/lib/libmath.b`. (The actual location may vary. It may be `/lib/libmath.b`.)

DIAGNOSTICS

If any file on the command line can not be opened, `bc` will report that the file is unavailable and terminate. Also, there are compile and run time diagnostics that should be self-explanatory.

BUGS

Error recovery is not very good yet.

AUTHOR

Philip A. Nelson
phil@cs.wwu.edu

ACKNOWLEDGEMENTS

The author would like to thank Steve Sommars (sesv@iwtsf.att.com) for his extensive help in testing the implementation. Many great suggestions were given. This is a much better product due to his involvement.

COPYRIGHT

NAME

`bined` – binary editor

SYNOPSIS

`bined` *file*

DESCRIPTION

`bined` is an interactive binary editor that can be used to modify existing binary files. Upon invocation, `bined` reads the entire file into memory and makes all changes to this separate image. Hence, as long as the changes have not been written to the file with the `esc-w` command, it is possible to quit without modifying the file.

OPERATION

`bined` writes a "page" of information at a time to the terminal. Each page consists of 20 lines of information. Each line contains a starting address (to the left) and 16 bytes displayed in both hex dump format and character format (to the right).

The file can be patched by modifying either the hex data or the character data. Whenever either is modified, the changes are immediately reflected in the other. When the cursor is on the hex side, any hex digit can be entered. When the cursor is on character side, any ASCII character can be entered. Control characters can be entered by pressing **`^V`** followed by the control character, like in `vi`. The following is a summary of the commands recognized by `bined`:

`Arrow keys`	These keys move the cursor around the current page, either on the hex side **`^h ^j`** or on the character side. If the cursor moved off the current page by pressing **`^k ^l`**, then any of these arrow keys, then the text will be scrolled up or down by one line as necessary, if possible.
`Tab`	Switch from hex side to character side and vice versa.
`^f`	Scroll forward one page.
`^b`	Scroll backward one page.
`^d`	Scroll up/forward half a page.

`^u`	Scroll down/backward half a page.
`Esc`	Enter command mode, where the following commands can be given:

- **`g`** Goto first page. (Beginning of file.)
- **`G`** Goto last page. (End of file.)
- **`q`** Quit without saving.
- **`x`** Save and Exit.
- **`w`** Write to file and continue.

After the commands **`q`**, **`x`**, and **`w`**, a carriage return is expected. At this time, backspace can be used to change the command, or two backspaces can be used to cancel the `ESC` mode. While in `ESC` mode, a "**`:`**" is displayed on the last line of the screen.

SEE ALSO

Utility Programs - `binfix, flook, od, show`

NAME

binfix – fix binary files

SYNOPSIS

binfix [*file* **]**

DESCRIPTION

binfix is a command-line-oriented file patching utility. All changes are performed immediately on the file.

OPERATION

Upon invocation, binfix displays a prompt at which various commands can be given. Commands can be abbreviated to any length. In case of ambiguity, the command that appears first in the help message will be taken. The following are the commands recognized by binfix:

help Displays the list of available commands.

file *filename* This command is used to specify a new file after patching the earlier one. The earlier file is closed and *filename* is opened. If there is an error opening *filename*, then binfix will be left with no currently open file, so that **show** or **fix** commands cannot be given until a successful **file** command is given.

show *offset length*

Displays length of bytes in hexadecimal starting from *offset* in the file. *Offset* must be given in hexadecimal without the "0x" prefix and *length* must be given in decimal. A "**.**" (dot) can be given for *offset*, in which case the current location in the file will be used.

fix *offset* **b|w|l**

Patch multiple consecutive bytes, shorts or longs, as specified by the second parameter. When this command is given, binfix will display the current byte, short or long value at the given *offset*, and will prompt for the new value to be entered. The new value is always expected in hexadecimal format without the "0x" prefix, and is written to the file immediately. After the value has been written, `binfix` will increment the offset by the appropriate amount (byte, short or long) and prompt for the next value. To discontinue patching, enter a "**.**" (dot) to get back to the command prompt. The values are always written in the native byte-ordering of the machine. If the file is created on a machine with opposite byte-ordering, then patching must be done byte-by-byte in the correct order.

done or **end** Closes the file and exits.

SEE ALSO

Utility Programs - `bined, flook, od, show`

NAME

`bison` – GNU parser generator (`yacc` replacement)

SYNOPSIS

bison [**-b** *file-prefix*] [**--file-prefix=***file-prefix*] [**-d**] [**--defines**] [**-l**] [**--no-lines**] [**-o** *outfile*] [**--output-file=***outfile*] [**-p** *prefix*] [**--name-prefix=***prefix*] [**-t**] [**--debug**] [**-v**] [**--verbose**] [**-V**] [**--version**] [**-y**] [**--yacc**] [**--fixed-output-files**] **file**

DESCRIPTION

`bison` is a parser generator in the style of `yacc`. It should be upwardly compatible with input files designed for `yacc`.

Input files should follow the `yacc` convention of ending in `.y`. Unlike `yacc`, the generated files do not have fixed names, but instead use the prefix of the input file. For instance, a grammar description file named `parse.y` would produce the generated parser in a file named `parse.tab.c`, instead of `yacc`'s `y.tab.c`.

This description of the options that can be given to `bison` is adapted from the node `Invocation` in the `bison.texinfo` manual, which should be taken as authoritative.

`bison` supports both traditional single-letter options and mnemonic long option names. Long option names are indicated with **--** instead of **-**. Abbreviations for option names are allowed as long as they are unique. When a long option takes an argument, like **--file-prefix**, connect the option name and the argument with **=**.

OPTIONS

-b *file-prefix*
--file-prefix=*file-prefix*

Specify a prefix to use for all `bison` output file names. The names are chosen as if the input file were named *file-prefix*`.c`.

-d
--defines Write an extra output file containing macro definitions for the token type names defined in the grammar and the semantic value type `YYSTYPE`, as well as a few `extern` variable declarations. If the parser output file is named *name*`.c` then this file is named *name*`.h`. This output file is essential if you wish to put the definition of `yylex` in a separate source file, because `yylex` needs to be able to refer to token type codes and the variable `yylval`.

-l
--no-lines Don't put any `#line` preprocessor commands in the parser file. Ordinarily `bison` puts them in the parser file so that the C compiler and debuggers will associate errors with your source file, the grammar file. This option causes them to associate errors with the parser file, treating it an independent source file in its own right.

-o *outfile*
--output-file=*outfile*
Specify the name *outfile* for the parser file. The other output files' names are constructed from *outfile* as described under the **-v** and **-d** switches.

-p *prefix*
--name-prefix=*prefix*
Rename the external symbols used in the parser so that they start with *prefix* instead of `yy`. The precise list of symbols renamed is `yyparse`, `yylex`, `yyerror`, `yylval`, `yychar`, and `yydebug`. For example, if you use `-p c`, the names become `cparse`, `clex`, and so on.

-t
--debug Output a definition of the macro `YYDEBUG` into the parser file, so that the debugging facilities are compiled.

-v
--verbose Write an extra output file containing verbose descriptions of the parser states and what is done for each type of look-ahead token in that state.

This file also describes all the conflicts, both those resolved by operator precedence and the unresolved ones.

The file's name is made by removing `.tab.c` or `.c` from the parser output file name, and adding `.output` instead.

Therefore, if the input file is `foo.y`, then the parser file is called `foo.tab.c` by default. As a consequence, the verbose output file is called `foo.output`.

-V
--version Print the version number of `bison`.

-y
--yacc
--fixed-output-files
Equivalent to `-o y.tab.c`; the parser output file is called `y.tab.c`, and the other outputs are called `y.output` and `y.tab.h`. The purpose of this switch is to imitate `yacc`'s output file name conventions. Thus, the following shell script can substitute for `yacc`:

bison –y $*

The long-named options can be introduced with **+** as well as **--**, for compatibility with previous releases. Eventually support for **+** will be removed, because it is incompatible with the POSIX.2 standard.

FILES

`/usr/lib/bison.simple`	simple parser
`/usr/lib/bison.hairy`	complicated parser

DIAGNOSTICS

Self explanatory.

COPYRIGHT

SEE ALSO

Utility Programs - `yacc`

NAME

`bm` – search a file for a string

SYNOPSIS

bm [*option*] ... [*string*] [*file...*]

DESCRIPTION

`bm` searches the input files (standard input default) for lines containing a particular string. Normally, each line found is copied to the standard output. `bm` strings are fixed sequences of characters: there are no wildcards, repetitions, or other features of regular expressions. `bm` is case sensitive. `bm` can be faster than `egrep`, and is attractive when searching for a fixed string that has embedded characters special to `egrep`.

OPTIONS

The following options are recognized.

-x	(Exact) only lines matched in their entirety are printed.
-l	The names of files with matching lines are listed (once) separated by newlines. (The matching lines are not printed.)
-c	Only a count of the number of matches is printed.
-e *string*	The pattern string is assumed to be the next argument after the **-e** flag. This allows strings beginning with "**-**".
-h	No filenames are printed, even if multiple files are searched.
-n	Each line is preceded by the number of characters from the beginning of the file to the match.
-s	Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.
-f *path*	The string list is taken from the path.

Unless the **-h** option is specified the file name is prepended to each output line if there is more than one input file. Care should be taken when using characters in the pattern string that are also meaningful to the shell. It is safest to quote the entire string.

`bm` searches for lines that contain one of the (newline-separated) `strings`, using the Boyer-Moore algorithm. It is far superior in terms of speed to the `grep` (`egrep`, `fgrep`) family of pattern matchers for fixed-pattern searching, and its speed increases with pattern length.

DIAGNOSTICS

Exit status is 0 if any matches are found; 1 if none; 2 for syntax errors or inaccessible files.

AUTHOR

Peter Bain, with modifications suggested by John Gilmore and Amir Plivatsky.

NOTE

Only 100 patterns are allowed.

Patterns may not contain newlines.

If a line (delimited by newlines, and the beginning and end of the file) is longer than 8000 characters (e.g. in a core dump), it will not be completely printed.

If multiple patterns are specified, the order of the output lines is not necessarily the same as the order of the input lines.

A line will be printed once for each different string on that line.

The algorithm cannot count lines.

The **-n** and **-c** options work differently from `fgrep`, the UNIX utility `bm` is intended to replace.

The **-v**, **-i**, and **-b** options (of `fgrep`) are not available.

SEE ALSO

Utility Programs - `egrep`

NAME

`bpconfig` – sets up participants in SCMP networks (PowerPC, 68k)

SYNOPSIS

bpconfig [-i *ifname***] [-n] [-f** *cfgfile***]**

bpconfig -r [-i *ifname***] [-n]**

DESCRIPTION

`bpconfig` sets up the configuration parameters for the MVME MPU modules sharing the VMEbus using SCMP. For each network participant `bpconfig` has to be invoked once before the SCMP network may be used. For proper operation of the SCMP network it is important that the `/etc/bplane.conf` file is identical on all systems involved.

The `bpconfig` utility is normally invoked at boot time by each SCMP peer. It uses the information contained in `/etc/bplane.conf` to update the configuration details of the MPU modules currently configured over the VMEbus. With no options, this utility takes `/etc/bplane.conf` as the database file and `bplane0` as the SCMP interface name.

Using the **-r** option the `bpconfig` utility may also be used to obtain the current configuration status of the SCMP network. For each SCMP participant the following information will be printed on standard output:

Type Boot-Flag DRAM-Address IO-Address IP-Address Name

The format of the output corresponds to the format of the `/etc/bplane.conf` configuration file. The network addresses are printed in IP dot-notation. The last field, *Name*, is printed only if the **-n** option is given.

OPTIONS

-f *cfgfile* Path name to the database file containing the configuration information to be set up for the SCMP system. The default is `/etc/bplane.conf`.

-i *ifname*	Different interface name for the SCMP interface. The default is `bplane0`.
-n	Show network addresses as numbers (normally `bpconfig` interprets addresses and attempts to display them symbolically).
-r	Read and print the current configuration parameters for all the MPU modules connected in an SCMP fashion.

FILES

`/etc/bplane.conf`

`/etc/config/netconfig.bplane`

`/sys/cfg/bplane.cfg`

SEE ALSO

File Formats – `bplane.conf`

NAME

`bYacc` – an LALR parser generator

SYNOPSIS

byacc[-dlrtv][-b *prefix* **]** *filename*

DESCRIPTION

`byacc` reads the grammar specification in the file *filename* and generates an LR parser for it. The parsers consist of a set of LALR parsing tables and a driver routine written in the C programming language. `byacc` normally writes the parse tables and the driver routine to the file `y.tab.c`.

The following options are available:

-b *prefix* The **-b** option changes the prefix prepended to the output file names to the string denoted by *prefix*. The default prefix is the character `y`.

-d The **-d** option causes the header file `y.tab.h` to be written.

-l If the **-l** option is not specified, `byacc` will insert `#line` directives in the generated code. The `#line` directives let the C compiler relate errors in the generated code to the user's original code. If the **-l** option is specified, `byacc` will not insert the `#line` directives. `#line` directives specified by the user will be retained.

-Q[y|n] '**-Qy**' instructs `byacc` to label the output file with the `byacc` version, using an `#ident` statement of the form

```
#ident "byacc: Berkeley Yacc (Cygnus)"
```

'**-Qn**' explicitly specifies that no such `#ident` statement should be included; this is the default.

-r The **-r** option causes `byacc` to produce separate files for code and tables. The code file is named `y.code.c`, and the tables file is named `y.tab.c`.

-t	The **-t** option changes the preprocessor directives generated by `byacc` so that debugging statements will be incorporated in the compiled code.
-V	Display version information for `byacc` on standard error.
-v	The **-v** option causes a human-readable description of the generated parser to be written to the file `y.output`.

If the environment variable `TMPDIR` is set, the string denoted by `TMPDIR` will be used as the name of the directory where the temporary files are created.

FILES

`y.code.c`

`y.tab.c`

`y.tab.h`

`y.output`

`/tmp/yacc.aXXXXXX`

`/tmp/yacc.tXXXXXX`

`/tmp/yacc.uXXXXXX`

DIAGNOSTICS

If there are rules that are never reduced, the number of such rules is reported on standard error. If there are any LALR() conflicts, the number of conflicts is reported on standard error.

NAME

`cal` – produce a calendar

SYNOPSIS

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

DESCRIPTION

`cal` produces a traditionally formatted calendar for the given *month* of the given *year*. If no parameters are given, the calendar for the current month is printed. If a single numeric argument is given, it is taken to be a *year*, and the calendar for that entire year is printed. If a single non-numeric argument is given, it is taken to be the name of a *month* (or some abbreviation thereof), and the calendar for the next occurrence of that month, this year or next, is printed.

If both arguments are given, the *month* may be specified by number (1 - 12) or by name.

The *year* is a number between 1 and 9999. Note that a 2-digit *year* such as 94 refers to a year before 100 A.D. and not a year in the 20th century.

SEE ALSO

Utility Programs - `date`

NAME

`calc` – simple desk calculator

SYNOPSIS

calc [*expression* **]**

DESCRIPTION

`calc` is a simple arithmetic calculator. If provided with a numeric *expression* (see below) as an argument, it computes the value and writes it to the standard output. If no argument is given, `calc` reads and evaluates expressions from standard input until end-of-file is detected.

`calc` performs computations with two data types: integer (32 bits) and floating point (64 bit double-precision). Types can be mixed, and `calc` will convert integer values to floating point as necessary. `calc` understands the following operators (precedence decreases from top to bottom):

~	Unary bitwise complement. Only usable with integers.
–	Unary arithmetic complement.
>>	Right shift
**	Exponentiation. (e.g., `2.0** 3 =23`)
*	Multiplication.
/	Division.
%	Remainder. Only usable with integers.
+	Addition.
–	Subtraction.
&	Bitwise `and`. Only for integers.
|	Bitwise `or`. Only for integers.
^	Bitwise `exclusive-or`. Only for integers.

=	Assignment. Left-hand side must be an identifier, whose name may consist of an arbitrary string of letters, digits, and underscores, beginning with a letter or underscore.
:	Integer width specifier. The left operand is an expression giving a width in bits for evaluation on the right-hand side. All integer computations on the right side are then carried out using two's-complement addition within the given number of bits.

Operands of the above operators (exceptions as noted) may be integer or floating-point constants, (using C syntax for scientific notation), variables, and built-in function calls. Integers may be given in octal by adding a leading `0`, in hexadecimal by adding a leading `0X` or `0x`, and in binary by adding a leading `0B` or `0b`. Variables which are referenced before values are assigned to them are created and given the value zero. The value of a variable may be an integer or a floating-point value; the variable is not bound permanently to either type. The variable named `A` is always set to the value of the most recent expression.

Several built-in functions are available. All are invoked by giving the name followed if necessary by a parenthesized argument list.

`acos(x)`	Arccosine.
`asin(x)`	Arcsine.
`atan(x)`	Arctangent.
`cos(x)`	Cosine.
`degrees`	Set angle mode to degrees. Arguments to trig functions will be automatically converted from degrees to radians, and results of inverse trig functions will be converted from radians to degrees.
`e`	The constant `e`, Euler's number.
`float(x)`	Convert integer argument to a floating point value.
`int(x)`	Convert floating point argument to an integer value.
`ln(x)`	Natural logarithm.

`log(x)`	Base 10 logarithm.
`mkdouble(x,y)`	Create a double-precision floating point value from the 64-bit pattern given as two integers.
`mkfloat(x)`	Create a single-precision floating point value from the 32-bit pattern given as a single integer.
`pi`	The constant `p`.
`radians`	Set angle mode to radians.
`sin(x)`	Sine.
`sqrt(x)`	Square root. Other roots can be obtained with the exponentiation operator.
`tan(x)`	Tangent.

Results of expressions are displayed in a manner appropriate to the data type. Integers are printed in decimal, hexadecimal, octal, and binary. The hexadecimal, octal, and binary versions are truncated to the number of bits of precision used for the outermost expression (normally 32). Floating point values are printed such that all significant digits are visible. The values of the exponent and mantissa fields of floating point numbers are also printed, in hexadecimal, for both single and double precision.

SEE ALSO

Utility Programs - `expr`

NAME

`cat` – concatenate files

SYNOPSIS

cat [-benstuv] [*filename ...* **]**

DESCRIPTION

`cat` concatenates the contents of one or more files onto the standard output. If no *filename* is given, or if the filename "**–**" is present in the list, `cat` will copy standard input to standard output. If standard out is not redirected to a file or through a pipe, the effect of `cat` is to print the files on the current `tty`.

The output of `cat` is line buffered if standard output is a `tty` device. Otherwise output is buffered in large blocks.

OPTIONS

–b Omits line numbers from blank lines; used with the **–n** option.

–e Appends a **$** character to each line; used with the **–v** option to indicate where the end-of-line character is.

–n Prepends the line number to each output line.

–s Replaces sequences of blank lines by a single blank line.

–t Converts tab characters to the string **^I**; used with the **–v** option.

–u Forces unbuffered (character-by-character) output.

–v Indicates positions of non-printing characters. Non-printable characters whose ascii value is less than 32 (40 octal) are output as **^**, followed by the value plus 64; thus ASCII 26 (control Z) is printed as **^Z**. The delete character, ASCII 127 (177 octal), is printed as **^?**. Characters whose value is greater than 127 are printed as **M–** followed by the printable version of the low 7 bits of the value. Note that printable control characters such as newline or tab are not translated.

NOTE

Commands like `cat a b > a` or `cat a b > b` do not work as expected and in fact will destroy information because the shell will open and reset the output files before `cat` has a chance to read from the input files.

SEE ALSO

Utility Programs - `cp, pr`

NAME

`cccp, cpp` – The GNU C-Compatible Compiler Preprocessor.

SYNOPSIS

cccp [-$] [-A *predicate***[(***value***)]] [-C] [-D** *name***[=***definition***]] [-dD] [-dM] [-I** *directory***] [-H] [-I-] [-imacros** *file***] [-include** *file***] [-idirafter** *dir***] [-iprefix** *prefix***] [-iwithprefix** *dir***] [-lang-c] [-lang-c++] [-lang-objc] [-lang-objc++] [-lint] [-M [-MG]] [-MM [-MG]] [-MD** *file* **] [-MMD** *file* **] [-nostdinc] [-nostdinc++] [-P] [-pedantic] [-pedantic-errors] [-traditional] [-trigraphs] [-U** *name***] [-undef] [-Wtrigraphs] [-Wcomment] [-Wall] [-Wtraditional]**
[*infile***|-] [** *outfile***|-]**

DESCRIPTION

The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. For a full explanation of the GNU C preprocessor, see the `info` file '`cpp.info`', or the manual *The C Preprocessor*. Both of these are built from the same documentation source file, '`cpp.texinfo`'. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options '**`-trigraphs`**', '**`-undef`**' and '**`-pedantic`**', but in practice the consequences of having strict ANSI Standard C make it undesirable to do this.

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful individually.

When you call the preprocessor individually, either name (`cpp` or `cccp`) will do—they are completely synonymous.

The C preprocessor expects two file names as arguments, *infile* and *outfile*. The preprocessor reads *infile* together with any other files it specifies with '`#include`'. All the output generated by the combined input files is written in *outfile*.

Either *infile* or *outfile* may be '**-**', which as *infile* means to read from standard input and as *outfile* means to write to standard output. Also, if *outfile* or both file names are omitted, the standard output and standard input are used for the omitted file names.

OPTIONS

Here is a table of command options accepted by the C preprocessor. These options can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

`-P` Inhibit generation of '`#`'-lines with line-number information in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the '`#`'-lines.

-C Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

-traditional Try to imitate the behavior of old-fashioned C, as opposed to ANSI C.

-trigraphs Process ANSI standard trigraph sequences. These are three-character sequences, all starting with '`??`', that are defined by ANSI C to stand for single characters. For example, `??/` stands for `\`, so `'??/n'` is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless '**-trigraphs**' is used, but if you ever notice the difference it will be with relief.

You don't want to know any more about trigraphs.

-pedantic Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows '`#else`' or '`#endif`'.

-pedantic-errors
Like '**-pedantic**', except that errors are produced rather than warnings.

-Wtrigraphs Warn if any trigraphs are encountered (assuming they are enabled).

-Wcomment
-Wcomments Warn whenever a comment-start sequence '`/*`' appears in a comment. (Both forms have the same effect).

-Wall Requests both '**-Wtrigraphs**' and '**-Wcomment**' (but not '**-Wtraditional**').

-Wtraditional Warn about certain constructs that behave differently in traditional and ANSI C.

-I *directory* Add the directory *directory* to the end of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '**-I**' option, the directories are scanned in left-to-right order; the standard system directories come after.

-I- Any directories specified with '**-I**' options before the '**-I-**' option are searched only for the case of '`#include "`*file*`"`'; they are not searched for '`#include <`*file*`>`'.

If additional directories are specified with '**-I**' options after the '**-I-**', these directories are searched for all '`#include`' directives.

In addition, the '**-I-**' option inhibits the use of the current directory as the first search directory for '`#include "`*file*`"`'. Therefore, the current directory is searched only if it is requested explicitly with '**-I.**'. Specifying both '**-I-**' and '**-I.**' allows you to control precisely which directories are searched before the current one and which are searched after.

-nostdinc Do not search the standard system directories for header files. Only the directories you have specified with '**-I**' options (and the current directory, if appropriate) are searched.

-nostdinc++ Do not search for header files in the C++ specific standard directories, but do still search the other standard directories. (This option is used when building `libg++`.)

-D *name* Predefine *name* as a macro, with definition '`1`'.

-D *name=definition*

Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one '**-D**' for the same *name*, the rightmost definition takes effect.

-U *name* — Do not predefine *name*. If both '**-U**' and '**-D**' are specified for one name, the '**-U**' beats the '**-D**' and the name is not predefined.

-undef — Do not predefine any nonstandard macros.

-A *name*(*value*) — Assert (in the same way as the `#assert` directive) the predicate *name* with tokenlist *value*. Remember to escape or quote the parentheses on shell command lines.

You can use '**-A-**' to disable all predefined assertions; it also undefines all predefined macros.

-dM — Instead of outputting the result of preprocessing, output a list of '`#define`' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor; assuming you have no file '`foo.h`', the command

```
touch foo.h; cpp -dM foo.h
```

will show the values of any predefined macros.

-dD — Like '**-dM**' except in two respects: it does *not* include the predefined macros, and it outputs *both* the '`#define`' directives and the result of preprocessing. Both kinds of output go to the standard output file.

-M [**-MG**] — Instead of outputting the result of preprocessing, output a rule suitable for `make` describing the dependencies of the main source file. The preprocessor outputs one `make` rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using '\\'-newline.

'**-MG**' says to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to '**-M**'.

This feature is used in automatic updating of makefiles.

-MM [**-MG**] Like '**-M**' but mention only the files included with '`#include "`*file*`"`'. System header files included with '`#include <`*file*`>`' are omitted.

-MD *file* Like '**-M**' but the dependency information is written to '*file*'. This is in addition to compiling the file as specified—'**-MD**' does not inhibit ordinary compilation the way '**-M**' does.

When invoking gcc, do not specify the '*file*' argument. `gcc` will create file names made by replacing '`.c`' with '`.d`' at the end of the input file names.

In Mach, you can use the utility `md` to merge multiple files into a single dependency file suitable for using with the '`make`' command.

-MMD *file* Like '**-MD**' except mention only user header files, not system header files.

-H Print the name of each header file used, in addition to other normal activities.

-imacros *file* Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '**-imacros** *file*' is to make the macros defined in *file* available for use in the main input. The preprocessor evaluates any '**-D**' and '**-U**' options on the command line before processing '**-imacros** *file*'.

-include *file* Process *file* as input, and include all the resulting output, before processing the regular input file.

-idirafter *dir* Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that '**-I**' adds to).

-iprefix *prefix* Specify *prefix* as the prefix for subsequent '**-iwithprefix**' options.

-iwithprefix *dir*
: Add a directory to the second include path. The directory's name is made by concatenating *prefix* and *dir*, where *prefix* was specified previously with '**-iprefix**'.

-lang-c
-lang-c++
-lang-objc
-lang-objc++
: Specify the source language. '**-lang-c++**' makes the preprocessor handle C++ comment syntax, and includes extra default include directories for C++, and '**-lang-objc**' enables the Objective C '`#import`' directive. '**-lang-c**' explicitly turns off both of these extensions, and '**-lang-objc++**' enables both.

  These options are generated by the compiler driver `gcc`, but not passed from the '`gcc`' command line.

-lint
: Look for commands to the program checker `lint` embedded in comments, and emit them preceded by '`#pragma lint`'. For example, the comment '`/* NOTREACHED */`' becomes '`#pragma lint NOTREACHED`'.

  This option is available only when you call `cpp` directly; `gcc` will not pass it from its command line.

-$
: Forbid the use of '`$`' in identifiers. This is required for ANSI conformance. `gcc` automatically supplies this option to the preprocessor if you specify '**-ansi**', but `gcc` doesn't recognize the '`-$`' option itself—to use it without the other effects of '**-ansi**', you must call the preprocessor directly.

SEE ALSO

Utility Programs	`cc, g++`
Documents -	‘`cpp`’ and ‘gcc’ entries in `info`
	The C Preprocessor, Richard M. Stallman.
	Using and Porting GNU CC (for version 2.0), Richard M. Stallman.

COPYING

NAME

`cd` – change current working directory

SYNOPSIS

cd *directory*

DESCRIPTION

`cd` makes *directory* the current working directory. The process running `cd` must have execute (search) permission over *directory*.

Since a new process is created to execute a program, `cd` is implemented as part of the shell rather than a separate program. Through the shell, the user may specify a list of directory paths for `cd` to search through to find *directory*. These paths are established in the shell variable `cdpath`. See `dlsh` documentation for more information.

SEE ALSO

Utility Programs - `dlsh, pwd`

System Calls - `chdir()`

Library Functions - `getwd()`

NAME

`cdwrite` – write a file or partition to a CD Recorder (x86)

SYNOPSIS

cdwrite -i {*sourcefile***|***raw_partition***} [-r] [-v] [-s** *speed***]** *cdrdevice*

DESCRIPTION

`cdwrite` writes a file or partition to a CD-R device. A CD-R is a CD Recorder that takes blank CD-R media. The media must be written to sequentially and at a specific rate depending on the speed factor. If the rate cannot be maintained, a data underflow occurs and the media is ruined. By not specifying the **-r** flag, you can do a test write to gain confidence that your system can meet the real-time deadline. After a simulation, you must eject and re-insert the media to reset the CD-R for the next write.

Two CD-R's are supported by `cdwrite`. They are the HP 4020i and the Yamaha CDR-100. `cdwrite` checks for exact matches in the inquiry string for these models. Since mode pages for various models of CD-R devices are not consistent, each model must have special code to work.

CD-R's have some internal buffering capability so slight stalls in the data stream can be tolerated. For example the HP 4020i has a 1 MB internal buffer. At speed factor 2 (300 KB/second) and assuming the write buffer is full, the HP 4020i can tolerate up to a 3.4 second interruption in the data flow. The Yamaha CDR-100 has an 512 KB buffer and supports 4x writing. With a full buffer at 4x, it can tolerate up to a 0.85 second interruption.

OPTIONS

-i *sourcefile*
-i *raw_partition* The source file to be written. This source file can be a regular file (*sourcefile*) or a raw partition (*raw_partition*).

-r Turn on the CD-R laser and do a real write. Without this option, the laser is in simulation mode and doesn't really write data to the media. Simulation mode is useful for ensuring the real-time deadlines can be met.

-v Verbose. Prints lots of useful information.

-s *speed*	Sets the speed factor used for writing the media. The default speed is 1. This means 150 KB a second data transfer rate. It takes 74 minutes to fill up a 650 MB CD at speed factor 1. Also, the approximate times to write lead-in and lead-out tracks are as follows:
	`-s 1` 4 minutes
	`-s 2` 2 minutes
	`-s 4` 1 minute
cdrdevice	The raw device name that represents the CD-R.

REQUIREMENTS

`cdwrite` is supported only on x86 systems. It requires at least 4 MB of free memory to run.

EXAMPLES

```
cdwrite -i /tmp/sourcefile -v -s 2 /dev/rsd2940.6
cdwrite -i /tmp/sourcefile -r -v -s 2 /dev/rsd2940.6
cdwrite -i /dev/rsd2940.0a -r -v -s 2 /dev/rsd2940.6
```

NOTE

If a data underflow occurs when using the **-r** flag, the media is useless.

BUGS

`cdwrite` will fail with an error message if the source partition is an IDE partition.

Some CD-ROM devices, usually older models, have trouble with CD-R media. The only solution is to purchase a new CD-ROM device.

NAME

`c++filt` – demangle C++ symbols

SYNOPSIS

c++filt [-_|--strip-underscores] [-s {gnu,lucid,arm} | --format={gnu,lucid,arm}] [*symbol...***]**

c++filt --help

c++filt --version

DESCRIPTION

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as *mangling*). The `c++filt` program does the inverse mapping: it decodes (*demangles*) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use `c++filt` to decipher individual symbols by specifying these symbols on the command line.

If no `symbol` arguments are given, `c++filt` reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

OPTIONS

-_
--strip-underscores

On some systems, both the C and C++ compilers put an underscore in front of every name. For example, the C name `foo` gets the low-level name `_foo`. This option removes the leading underscore.

-s `{gnu,lucid,arm}`
--format=`{gnu,lucid,arm}`

GNU `nm` can decode three different methods of mangling, used by different C++ compilers. This option selects which method it uses: the one used by the GNU compiler, the one used by the Lucid compiler, or the one specified by the *C++ Annotated Reference Manual*. The default is the GNU style.

--help Print a summary of the options to `c++filt` and exit.

--version Print the version number of `c++filt` and exit.

SEE ALSO

Documents - ‘`binutils`’ entry in `info`

The GNU Binary Utilities, Roland H. Pesch.

COPYING

NAME

`chgrp` – change group ownership of files

SYNOPSIS

`chgrp [ -f ]` *group file ...*

DESCRIPTION

`chgrp` changes the group ownership of the named *files* to *group*. The new *group* can be given as a group name or a numeric group ID.

Unless the current real (not effective) user ID is zero (the super user), `chgrp` will only allow changes to files whose owning group ID is one of those in the current group list; the new *group* must be in the current group list as well.

OPTIONS

`-f`	Suppresses error messages in case of failure.

FILES

`/etc/passwd`	User ID information
`/etc/group`	Group names

SEE ALSO

Utility Programs - `chmod, chown, ls, shownode`

System Calls - `chown(), getgroups()`

NAME

`chmod` – change access modes of a file

SYNOPSIS

chmod [-R] [-d *dirmode***] [-v]** *mode file...*

DESCRIPTION

`chmod` is used to change the permission mode of a file or directory. The mode of each specified file is changed according to the mode argument. The mode can be specified in either an absolute or symbolic way.

An absolute mode is an octal integer created by a OR'ing together one or more of the following values (expressed in octal):

`1000`	Set sticky bit. This bit, when set, causes the program-text segment of the file to stay in memory even if no process is using it.
`0400`	Set owner read permission.
`0200`	Set owner write permission.
`0100`	Set owner execute permission.
`0040`	Set group read permission.
`0020`	Set group write permission.
`0010`	Set group execute permission.
`0004`	Set other read permission.
`0002`	Set other write permission.
`0001`	Set other execute permission.

A symbolic mode has the form:

```
[agou] [+-=] [rwxsgou] [, ... ]
```

The components of the symbolic form are:

a	Owner, group, and other access.

g	Group access.
o	Other access.
u	Owner access.
+	Adds specified permissions to the group, other, or owner category of the files specified.
–	Subtracts specified permissions from the group, other, or owner category of the files specified.
=	Sets the specified permissions to the group, other, or owner category of the files specified.
r	Read permission.
w	Write permission.
x	Execute permission.
s	Set user or group ID when executed.

A list of symbolic "who-how-what" mode changes can be given, separated by commas.

OPTIONS

–R	Recursively change mode on all files in each directory *file*.
–d *dirmode*	Use *dirmode* instead of mode for each file which is a directory. Applies to files explicitly listed and to files encountered during a recursive `chmod`.
–v	Print list of files encountered during recursive `chmod`.

EXAMPLES

`chmod u–w xxx`	Removes permission for the owner to write to file `xxx`.
`chmod +x yyy`	Adds execute permission to the file `yyy`. Because no "who" field is explicitly given, `chmod` assumes **a** but also modifies the resultant mask according to the current *umask* value.

`chmod u+rw,o-w prog`
Adds read and write permissions for the owner, and removes write permission for others, from the file `prog`.

SEE ALSO

Utility Programs - `chown, ls, shownode`

System Calls - `chmod(), stat()`

NAME

`chown` – change file ownership

SYNOPSIS

`chown [-f]` *user file ...*

DESCRIPTION

`chown` changes the user ID number associated with each file to the user ID given. The new owner can be specified as a decimal number or as a name from the password file.

If the **`-f`** option is given `chown` will not report errors encountered.

NOTE

`chown` is only available to the root user (user ID 0). Attempts by other users to change file ownership will result in an error message.

SEE ALSO

Utility Programs - `chmod, ls, shownode`

System Calls - `chown(), stat()`

NAME

`chroot` – change root directory

SYNOPSIS

`chroot` *newroot* **[** *command* **]**

DESCRIPTION

The `chroot` command changes its root directory to the supplied directory newroot and exec's command, if supplied, or an interactive copy of your shell.

Note, *command* or the shell are run as your real-user-id.

ENVIRONMENT

The following environment variable is referenced by `chroot`:

`SHELL`	If set, the string specified by `SHELL` is interpreted as the name of the shell to `exec`. If the variable `SHELL` is not set, `/bin/sh` is used.

ACKNOWLEDGEMENTS

This section was developed by the University of California, Berkeley.

SEE ALSO

System Calls - `chdir`

NAME

`ci` – check in RCS revisions

SYNOPSIS

ci [*options* **]** *file ...*

DESCRIPTION

`ci` stores new revisions into RCS files. Each file name ending in "`,v`" is taken to be an RCS file, all others are assumed to be working files containing new revisions. `ci` deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of `co`).

1. Both the RCS file and the working file are given. The RCS file name is of the form `path1/workfile,v` and the working file name is of the form `path2/workfile`, where `path1/` and `path2/` are (possibly different or empty) paths and `workfile` is a file name.
2. Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing `path1/` and the suffix "`,v`".
3. Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing `path2/` and appending the suffix "`,v`". If the RCS file is omitted or specified without a path, then `ci` looks in the current directory for a file called `RCSDIR`. If it exists, this file is taken as a list of paths (one per line) where the desired RCS file might be found. If `RCSDIR` does not exist, or if the RCS file is not found in any of the directories named therein, then `ci` looks for a subdirectory called RCS in the current directory and searches for the RCS file there.

For `ci` to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This restriction is not enforced for the owner of the file, unless locking is set to strict (see `rcs`). A lock held by someone else may be broken with the `rcs` command.

Normally, `ci` checks whether the revision to be deposited is different from the preceding one. If it is not different, `ci` either aborts the deposit (if **-q** is given) or asks whether to abort (if **-q** is omitted). A deposit can be forced with the **-f** option.

For each revision deposited, `ci` prompts for a log message. The log message should summarize the change and must be terminated with a line containing a single "." or a control-D. If several files are checked in, `ci` asks whether to reuse the previous log message. If the `stdin` is not a terminal, `ci` suppresses the prompt and uses the same log message for all files. See also **-m**.

The number of the deposited revision can be given by any of the options **-r**, **-f**, **-k**, **-l**, **-u**, or **-q** (see **-r**).

If the `RCS` file does not exist, `ci` creates it and deposits the contents of the working file as the initial revision (default number: 1.1). The access list is initialized to empty. Instead of the log message, `ci` requests descriptive text (see **-t** below).

OPTIONS

-r *rev* Assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default. If *rev* is omitted, `ci` derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller holds no lock, but he is the owner of the file and locking is not set to strict, then the revision is appended to the trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch. If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch.

If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can be appended to the end, but not inserted.

-f *rev* Forces a deposit; the new revision is deposited even it is not different from the preceding one.

-k *rev* Searches the working file for keyword values to determine its revision number, creation date, author, and state (see `co`), and assigns these values to the deposited revision, rather than computing them locally. A revision number given by a command option overrides the number in the working file. This option is useful for software distribution. A revision that is sent to several sites should be checked in with the **-k** option at these sites to preserve its original number, date, author, and state.

-l *rev* Works like **-r**, except it performs an additional locked check-out with for the deposited revision. Thus, the deposited revision is immediately checked out again and locked. This is useful for saving a revision although one wants to continue editing it after the checkin.

-u *rev* Works like **-l**, except that the deposited revision is not locked. This is useful if one wants to process (e.g., compile) the revision immediately after checkin.

-q *rev* Quiet mode; diagnostic output is not printed. A revision that is not different from the preceding one is not deposited, unless **-f** is given.

-m *msg* Uses the string *msg* as the log message for all revisions checked in.

-n *name* — Assigns the symbolic name *name* to the number of the checked-in revision. `ci` prints an error message if *name* is already assigned to another number.

-N *name* — Same as **-n**, except that it overrides a previous assignment of *name*.

-s *state* — Sets the state of the checked-in revision to the identifier *state*. The default is "`Exp`".

-t *txtfile* — Writes descriptive text into the `RCS` file (deletes the existing text). If *txtfile* is omitted, `ci` prompts the user for text supplied from the `stdin`, terminated with a line containing a single "." or control-D. Otherwise, the descriptive text is copied from the file *txtfile*. During initialization, descriptive text is requested even if **-t** is not given. The prompt is suppressed if standard input is not a terminal.

FILE CREATION DETAILS

An `RCS` file created by `ci` inherits the read and execute permissions from the working file. If the `RCS` file exists already, `ci` preserves its read and execute permissions. `ci` always turns off all write permissions of `RCS` files.

The caller of the command must have read/write permission for the directories containing the `RCS` file and the working file, and read permission for the `RCS` file itself. A number of temporary files are created. A semaphore file is created in the directory containing the `RCS` file. `ci` always creates a new `RCS` file and unlinks the old one. This strategy makes links to `RCS` files useless.

DIAGNOSTICS

For each revision, `ci` prints the `RCS` file, the working file, and the number of both the deposited and the preceding revision. The exit status always refers to the last file checked in, and is 0 if the operation was successful, 1 otherwise.

COPYRIGHT NOTICE

The RCS system is the work of Walter F. Tichy of Purdue University. It was obtained by Lynx from the Free Software Foundation. As such, Lynx Real-Time Systems will make available for nominal postage and materials cost complete source code for all parts of the RCS system. Furthermore, LynxOS users are hereby permitted to copy and distribute RCS at will under these same terms; that is, notice that source is available from Lynx Real-Time Systems must be given along with any RCS distribution, or else the source itself. If any RCS source is to be used as a component of some other software system, the Free Software Foundation should be contacted for details:

Free Software Foundation, Inc.
675 Massachusetts Ave.
Cambridge, MA 02139

Note that RCS as distributed by FSF has been modified in small ways by Lynx Real-Time Systems, Inc., and thus although source for RCS may be available elsewhere it will be different from LynxOS RCS. Lynx Real-Time Systems assumes no responsibility for copies of RCS obtained elsewhere.

SEE ALSO

Utility Programs - `co, ident, rcs, rcsdiff, rcsintro, rlog`

File Formats - `rcsfile`

NAME

`clear` – clear terminal screen

SYNOPSIS

`clear`

DESCRIPTION

`clear` clears the terminal screen, if possible. It determines the terminal type from the environment variable `TERM` and then uses `/etc/termcap` to figure out how to clear the screen. `clear` is actually a special case of the `termcap` command.

FILES

`/etc/termcap` terminal capability data base

SEE ALSO

Utility Programs - `termcap`

NAME

`cmp` – compare two files

SYNOPSIS

cmp [**-cls**] [**--show-chars**] [**--verbose**] [**--silent**] [**--quiet**] *file1* [*file2*]

DESCRIPTION

This manual page documents the GNU version of `cmp`. `cmp` compares the two given files, and if they differ, tells the first byte and line number where they differ. The file name – is always the standard input. If one file name is omitted, the standard input is used as well.

It exits with the value

`0`	when the files are identical.
`1`	when the files differ.
`2`	when an error has occurred.

OPTIONS

-c
--show-chars Print the differing characters like `cat` **-t** (except that newlines are printed as `^J`).

-l
--verbose Print the (decimal) offsets and (octal) values of all differing bytes.

-s
--silent
--quiet Do not print anything; only return an exit status indicating whether the files differ.

NAME

co – check out RCS revisions

SYNOPSIS

co [*options* **]** *file ...*

DESCRIPTION

co retrieves revisions from RCS files. Each file name ending in ",v" is taken to be an RCS file. All other files are assumed to be working files. co retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1. Both the RCS file and the working file are given. The RCS file name is of the form path1/workfile,v and the working file name is of the form path2/workfile, where path1/ and path2/ are (possibly different or empty) paths and workfile is a file name.
2. Only the RCS file is given. Then the working file is assumed to be in the current directory and its name is derived from the name of the RCS file by removing path1/ and the suffix ",v".
3. Only the working file is given. Then the name of the RCS file is derived from the name of the working file by removing path2/ and appending the suffix ",v".

If the RCS file is omitted or specified without a path, then co looks in the current directory for a file called RCSDIR. If it exists, this file is taken as a list of paths (one per line) where the desired RCS file might be found. If RCSDIR does not exist, or if the RCS file is not found in any of the directories named therein, then co looks for a subdirectory called RCS in the current directory and searches for the RCS file there.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision prevents overlapping updates. A revision checked out for reading or processing (e.g., compiling) need not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the `rcs` command.) `co` with locking requires the caller to be on the access list of the `RCS` file, unless he is the owner of the file or the superuser, or the access list is empty. `co` without locking is not subject to access list restrictions.

A revision is selected by number, checkin date/time, author, or state. If none of these options are specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. The options for date/time, author, and state retrieve a revision on the selected branch. The "selected branch" is either derived from the revision number (if given), or is the highest branch on the trunk. A revision number may be attached to one of the options **-l**, **-p**, **-q**, or **-r**.

A `co` command applied to an `RCS` file with no revisions creates a zero-length file. `co` always performs keyword substitution (see below).

OPTIONS

-l *rev* Locks the checked out revision for the caller. If omitted, the checked out revision is not locked. See option **-r** for handling of the revision number *rev*.

-p *rev* Prints the retrieved revision on the standard output rather than storing it in the working file. This option is useful when `co` is part of a pipe.

-q *rev* Quiet mode; diagnostics are not printed.

-d *date* Retrieves the latest revision on the selected branch whose checkin date/time is less than or equal to *date*. The date and time may be given in free format and are converted to local time. Examples of formats for *date*:

```
22-April-1982, 17:20-CDT
2:25 AM, Dec. 29, 1983
Tue-PDT, 1981, 4pm Jul 21
Fri, April 16 15:52:25 EST 1982
```

Most fields in the date and time may be defaulted. `co` determines the defaults in the order year, month, day, hour, minute, and second (most to least significant). At least one of these fields must be provided. For omitted fields that are of higher significance than the highest provided field, the current values are assumed. For all other omitted fields, the lowest possible values are assumed. For example, the date `20, 10:30` defaults to `10:30:00` of the 20th of the current month and current year. The date/time must be quoted if it contains spaces.

-r *rev* Retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *rev* is composed of one or more numeric or symbolic fields separated by "**.**". The numeric equivalent of a symbolic field is specified with the **-n** option of the commands `ci` and `rcs`.

-s *state* Retrieves the latest revision on the selected branch whose state is set to *state*.

-w *login* Retrieves the latest revision on the selected branch which was checked in by the user with login name *login*. If the argument *login* is omitted, the caller's login is assumed.

-j *joinlist* Generates a new revision which is the `join` of the revisions on *joinlist*. *joinlist* is a comma-separated list of pairs of the form `rev2:rev3`, where `rev2` and `rev3` are (symbolic or numeric) revision numbers. For the initial such pair, `rev1` denotes the revision selected by the options **-l**, ..., **-w**. For all other pairs, `rev1` denotes the revision generated by the previous pair. (Thus, the output of one `join` becomes the input to the next.)

For each pair, `co` joins revisions `rev1` and `rev3` with respect to `rev2`. This means that all changes that transform `rev2` into `rev1` are applied to a copy of `rev3`. This is particularly useful if `rev1` and `rev3` are the ends of two branches that have `rev2` as a common ancestor. If `rev1` **|** `rev2` **|** `rev3` on the same branch, joining generates a new revision which is like `rev3`, but with all changes that lead from `rev1` to `rev2` undone. If changes from `rev2` to `rev1` overlap with changes from `rev2` to `rev3`, `co` prints a warning and includes the overlapping sections, delimited as follows:

```
<<<<<<< rev1
=======
>>>>>>> rev3
```

For the initial pair, `rev2` may be omitted. The default is the common ancestor. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option **-l** is present, the initial `rev1` is locked.

KEYWORD SUBSTITUTION

Strings of the form `$keyword$` and `$keyword:...$` embedded in the text are replaced with strings of the form `$keyword: value $`, where keyword and value are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form `$keyword$`. On checkout, `co` replaces these strings with strings of the form `$keyword: value $`. If a revision containing strings of the latter form is checked back in, the value fields will be replaced during the next checkout. Thus, the keyword values are automatically updated on checkout.

Keywords and their corresponding values (must be specified in the form $*keyword*$):

`Author`	The login name of the user who checked in the revision.
`Date`	The date and time the revision was checked in.
`Header`	A standard header containing the `RCS` file name, the revision number, the date, the author, and the state.
`Locker`	The login name of the user who locked the revision (empty if not locked).
`Log`	The log message supplied during checkin, preceded by a header containing the `RCS` file name, the revision number, the author, and the date. Existing log messages are *not* replaced. Instead, the new log

message is inserted after `$<Log-keyword>:...$`. This is useful for accumulating a complete change log in a source file.

`Revision`	The revision number assigned to the revision.
`Source`	The full pathname of the `RCS` file.
`State`	The state assigned to the revision with `rcs -s` or `ci -s`.

DIAGNOSTICS

The `RCS` file name, the working file name, and the revision number retrieved are written to the diagnostic output. The exit status always refers to the last file checked out, and is 0 if the operation was successful, 1 otherwise.

EXAMPLES

Suppose the current directory contains a subdirectory `RCS` with an `RCS` file `io.c,v`. Then all of the following commands retrieve the latest revision from `RCS/io.c,v` and store it into `io.c`.

```
co io.c; co RCS/io.c,v; co io.c,v;
co io.c RCS/io.c,v; co io.c io.c,v;
co RCS/io.c,v io.c; co io.c,v io.c;
```

FILE CREATION DETAILS

The working file inherits the read and execute permissions from the `RCS` file. In addition, the owner write permission is turned on, unless the file is checked out unlocked and locking is set to `strict` (see `rcs`).

If a file with the name of the working file exists already and has write permission, `co` aborts the checkout if **-q** is given, or asks whether to abort if **-q** is not given. If the existing working file is not writable, it is deleted before the checkout.

The caller of the command must have write permission in the working directory, read permission for the `RCS` file, and either read permission (for reading) or read/write permission (for locking) in the directory which contains the `RCS` file.

A number of temporary files are created. A semaphore file is created in the directory of the `RCS` file to prevent simultaneous update.

NOTE

The option **`-d`** gets confused in some circumstances, and accepts no date before 1970. There is no way to suppress the expansion of keywords, except by writing them differently. The option **`-j`** does not work for files that contain lines with a single "`.`".

COPYRIGHT NOTICE

SEE ALSO

Utility Programs - `ci, ident, rcs, rcsdiff, rcsintro, rlog`

File Formats - `rcsfile`

NAME

`comm` – select or reject lines common to two sorted files

SYNOPSIS

`comm [-123]` *file1 file2*

`comm --help`

`comm --version`

DESCRIPTION

`comm` prints lines that are common, and lines that are unique, to two input files. The two files must be sorted before `comm` can be used. The file name '**–**' means the standard input.

With no options, `comm` produces three column output. Column one contains lines unique to *file1*, column two contains lines unique to *file2*, and column three contains lines common to both files.

OPTIONS

`-1`	Omit column one (lines unique to *file1*).
`-2`	Omit column two (lines unique to *file2*).
`-3`	Omit column three (lines common to both files).
`--help`	Print a usage message and exit with a status code indicating success.
`--version`	Print version information on standard output then exit.

SEE ALSO

Utility Programs- `cmp, diff, sort, uniq`

NAME

`compress, uncompress,`
`zcat –` compress and uncompress files

SYNOPSIS

compress [-cdFfnqv] [-b *maxbits* **] [** *file ...* **]**

uncompress [-cFfnqv] [*file ...* **]**

zcat [*file ...* **]**

DESCRIPTION

Compresses the specified files or standard input. Each *file* is replaced by a new file whose name is formed by adding "`.Z`" to the end of the original name. If no files are specified, the compression is applied to the standard input and is written to standard output regardless of the results. If an attempt is made to compress a file and the result is actually larger than the original, no new file is created; the original is left alone. Compressed files can be restored to their original form by specifying the **-d** option, or by running `uncompress` (actually a link to `compress`), on the "`.Z`" files or the standard input.

If the output file exists, it will not be overwritten unless the **-f** or **-F** flag is given. If neither of these is specified and `compress` is run in the foreground, the user is prompted as to whether the file should be overwritten.

If the **-F** or the **-f** flag is given, all files specified are replaced with `.Z` files—even if the file didn't get smaller.

When file names are given, the ownership (if run by `root`), modes, accessed and modified times are maintained between the file and its `.Z` version. In this respect, `compress` can be used for archival purposes, yet can still be used with make after uncompression.

The **-c** option causes the results of the `compress/uncompress` operation to be written to `stdout`; no files are changed. Running the `zcat` program is the same as specifying **-c** to `uncompress` (all files are unpacked and written to `stdout`). `compress` uses the modified Lempel-Ziv algorithm described 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 *maxbits* limit as specified by the **-b** flag is reached (default 16). *maxbits* 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 *maxbits* limit is reached, `compress` periodically checks the compression ratio. If it is increasing, `compress` continues to use the codes that were previously found in the file. 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.

A two byte magic number is prepended to the file to ensure that neither uncompression of random text nor recompression of compressed text are attempted. In addition, the *maxbits* specified during `compress` is written to the file so that the **-b** flag can be omitted for `uncompress`. The **-n** flag should be used to `uncompress` older files that do not have the magic number.

The amount of compression obtained depends on the size of the input file, the amount of *maxbits* per code, and the distribution of character substrings. Typically, text files, such as C programs, are reduced by 50-60%. Compression is generally much better than that achieved by Huffman coding or adaptive Huffman coding and takes less time to compute.

DIAGNOSTICS

After each file is compressed, a message may be printed giving the percentage of the input file that has been saved by compression. This message is only printed when the **-v** (verbose) flag is given. This message is suppressed when the **-q** (quiet) flag is given. The default is the quiet mode.

The exit status is normally 0; if the last file gets bigger after compression, the exit status is 2; if an error occurs, the exit status is 1.

COPYRIGHT NOTICE

This source code for `compress` is available upon request.

NAME

`config` – produce system configuration tables

SYNOPSIS

`config` *table* **[** *conf* **[** *nodetab* **[** *sysdev* **]]]**

DESCRIPTION

`config` reads a conveniently handled *table* describing the operating system configuration and produces three files used in actually creating LynxOS. The files created will be named *conf*, *nodetab*, and *sysdev*; if any of these are omitted, the contents are written to standard output. A configuration primarily involves the selection of device drivers and devices to be linked into the executable image of the system.

`conf` is an include file which becomes part of `conf.c`. It includes the structure definitions for device drivers and devices. *nodetab* is a table of node declarations for `mknod` which reflects the major and minor numbers appropriate for the configuration. *sysdev* contains any necessary definitions for device numbers used internally by the operating system; the only such device currently is the console serial port.

NOTE

Although `config` will write to any file, for purposes of creating the OS with the supplied `makefile` and objects the following names are used:

conf	`CONFIG.h`
nodetab	`nodetab`
sysdev	`sysdev.h`

The configuration file itself is usually called `CONFIG.TBL`.

SEE ALSO

File Formats - `config`

NAME

`cp` – copy files

SYNOPSIS

cp [-irv] *file1 file2*

cp [-irv] *file ... directory*

DESCRIPTION

`cp` creates copies of files. The destination file (*file2*) is created if it does not exist, or overwritten if it does. If more than two files are given, then the last filename must be a directory, and the files are copied into it with their original base names. `cp` will not copy a file to itself.

OPTIONS

-i Instructs `cp` to prompt the user whenever a file is about to be overwritten. Answering **y** causes `cp` to continue; anything else causes `cp` to skip that copy operation.

-r Causes `cp` to recursively copy all subtrees of all `file` arguments which are directories and deposit them into the destination directory. `cp` will flag an error if the destination file is not a directory. Hard and symbolic links are not understood by `cp`; a complete copy will be made based upon each directory entry in the source subtree. To truly duplicate subtrees, complete with link information, see `tar`.

-v Print list of files encountered during a recursive `cp`.

EXAMPLES

This will do a simple copy from *file1* into *file2*.

```
cp file1 file2
```

This will copy *file1*,*file2* and *file3* into the directory *dir1*.

```
cp file1 file2 file3 dir1
```

This will copy all files recursively in *dir1* into *dir2*; it will also copy *file1* into *dir2*.

```
cp -r dir1 file1 dir2
```

SEE ALSO

Utility Programs - `cat, pr, mv`

NAME

cpio – copy files to and from archives

SYNOPSIS

cpio {-o|--create} [-0acvABLV] [-C *bytes***] [-H** *format***]**
[-M *message***] [-O [[***user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***]**
[--file=[[*user***@]***host***:]***archive***] [--format=***format***]**
[--message=*message***] [--null] [--reset-access-time]**
[--verbose] [--dot] [--append] [--block-size=*blocks***]**
[--dereference] [--io-size=*bytes***]**
[--force-local] < *name-list* **[>** *archive***]**

cpio {-i|--extract} [-bcdfmnrtsuvBSV] [-C *bytes***] [-E** *file***]**
[-H *format***] [-M** *message***] [-R [***user***][:.][***group***]]**
[-I [[*user***@]***host***:]***archive***] [-F [[***user***@]***host***:]***archive***]**
[--file=[[*user***@]***host***:]***archive***]**
[--make-directories] [--nonmatching]
[--preserve-modification-time] [--numeric-uid-gid]
[--rename] [--list] [--swap-bytes] [--swap] [--dot]
[--unconditional] [--verbose] [--block-size=*blocks***]**
[--swap-halfwords] [--io-size=*bytes***] [--pattern-file=***file***]**
[--format=*format***] [--owner=[***user***][:.][***group***]]**
[--no-preserve-owner] [--message=*message***]**
[--force-local] [*pattern*...**] [<** *archive***]**

cpio {-p|--pass-through} [-0adlmuvLV] [-R [*user***][:.][***group***]]**
[--null] [--reset-access-time] [--make-directories] [--link]
[--preserve-modification-time] [--unconditional] [--verbose]
[--dot] [--dereference] [--owner=[*user***][:.][***group***]]**
[--no-preserve-owner] *destination-directory* **<** *name-list*

cpio --version

DESCRIPTION

This manual page documents the GNU version of `cpio`. `cpio` copies files into or out of a `cpio` or `tar` archive, which is a file that contains other files plus information about them, such as their pathname, owner, time stamps, and access permissions. The archive can be another file on the disk, a magnetic tape, or a pipe. `cpio` has three operating modes.

In copy-out mode, `cpio` copies files into an archive. It reads a list of file names, one per line, on the standard input, and writes the archive onto the standard output. A typical way to generate the list of file names is with the `find` command; you should give `find` the `-depth` option to minimize problems with permissions on directories that are unwritable or not searchable.

In copy-in mode, `cpio` copies files out of an archive or lists the archive contents. It reads the archive from the standard input. Any non-option command line arguments are shell globbing patterns; only files in the archive whose names match one or more of those patterns are copied from the archive. Unlike in the shell, an initial `.` in a file name does match a wildcard at the start of a pattern, and a `/` in a file name can match wildcards. If no patterns are given, all files are extracted.

In copy-pass mode, `cpio` copies files from one directory tree to another, combining the copy-out and copy-in steps without actually using an archive. It reads the list of files to copy from the standard input; the directory into which it will copy them is given as a non-option argument.

`cpio` supports the following archive formats: binary, old ASCII, new ASCII, crc, old tar, and POSIX.1 tar. The binary format is obsolete because it encodes information about the files in a way that is not portable between different machine architectures. The old ASCII format is portable between different machine architectures, but should not be used on file systems with more than 65536 i-nodes. The new ASCII format is portable between different machine architectures and can be used on any size file system, but is not supported by all versions of `cpio`; currently, it is only supported by GNU and Unix System V R4. The crc format is like the new ASCII format, but also contains a checksum for each file which `cpio` calculates when creating an archive and verifies when the file is extracted from the archive.

The tar format is provided for compatibility with the `tar` program. It can not be used to archive files with names longer than 100 characters, and can not be used to archive “special” (block or character devices) files. The POSIX.1 tar format can not be used to archive files with names longer than 255 characters (less unless they have a `/` in just the right place).

By default, `cpio` creates binary format archives, for compatibility with older `cpio` programs. When extracting from archives, `cpio` automatically recognizes which kind of archive it is reading and can read archives created on machines with a different byte-order.

Some of the options to `cpio` apply only to certain operating modes; see SYNOPSIS section for a list of which options are allowed in which modes.

OPTIONS

-0, --null In copy-out and copy-pass modes, read a list of file names terminated by a null character instead of a newline, so that files whose names contain newlines can be archived. GNU `find` is one way to produce a list of null-terminated file names.

-a, --reset-access-time
Reset the access times of files after reading them, so that it does not look like they have just been read.

-A, --append Append to an existing archive. Only works in copy-out mode. The archive must be a disk file specified with the **-O** or **-F** *(—file)* option.

-b, --swap In copy-in mode, swap both half-words of words and bytes of half-words in the data. Equivalent to **-sS**. Use this option to convert 32-bit integers between big-endian and little-endian machines.

-B Set the I/O block size to 5120 bytes. Initially the block size is 512 bytes.

--block-size=*blocks*
Set the I/O block size to *blocks* * 512 bytes.

-c Use the old portable (ASCII) archive format.

-C *bytes*, **--io-size=***bytes*
Set the I/O block size to *bytes* bytes.

-d, --make-directories
Create leading directories where needed.

-E *file*, **--pattern-file**=*file*
In copy-in mode, read additional patterns specifying file names to extract or list from *file*. The lines of *file* are treated as if they had been non-option arguments to `cpio`.

-f, --nonmatching
Only copy files that do not match any of the given patterns.

-F, --file=*archive*
Archive file name to use instead of standard input or output. To use a tape drive on another machine as the archive, use a file name that starts with `HOSTNAME:`. The hostname can be preceded by a username and an `@` to access the remote tape drive as that user, if you have permission to do so (typically an entry in that user's `~/.rhosts` file).

--force-local With **-F**, **-I**, or **-O**, take the archive file name to be a local file even if it contains a colon, which would ordinarily indicate a remote host name.

-H *format*, **--format**=*format*
Use archive format *format*. The valid formats are listed below; the same names are also recognized in all-caps. The default in copy-in mode is to automatically detect the archive format, and in copy-out mode is `bin`.

`bin`	The obsolete binary format.
`odc`	The old (POSIX.1) portable format.
`newc`	The new (SVR4) portable format, which supports file systems having more than 65536 i-nodes.
`crc`	The new (SVR4) portable format with a checksum added.
`tar`	The old tar format.
`ustar`	The POSIX.1 tar format. Also recognizes GNU `tar` archives, which are similar but not identical.

-i, --extract Run in copy-in mode.

-I *archive* Archive file name to use instead of standard input. To use a tape drive on another machine as the archive, use a file name that starts with `HOSTNAME:`. The hostname can be preceded by a username and an `@` to access the remote tape drive as that user, if you have permission to do so (typically an entry in that user's `~/.rhosts` file).

-k Ignored; for compatibility with other versions of **cpio**.

-l, --link Link files instead of copying them, when possible.

-L, --dereference
Dereference symbolic links (copy the files that they point to instead of copying the links).

-m, --preserve-modification-time
Retain previous file modification times when creating files.

-M *message*, **--message**=*message*
Print *message* when the end of a volume of the backup media (such as a tape or a floppy disk) is reached, to prompt the user to insert a new volume. If *message* contains the string `%d`, it is replaced by the current volume number (starting at 1).

-n, --numeric-uid-gid
In the verbose table of contents listing, show numeric `UID` and `GID` instead of translating them into names.

--no-preserve-owner
In copy-in mode and copy-pass mode, do not change the ownership of the files; leave them owned by the user extracting them. This is the default for non-root users, so that users on System V don't inadvertently give away files.

-o, --create Run in copy-out mode.

-O *archive* Archive file name to use instead of standard output. To use a tape drive on another machine as the archive, use a file name that starts with `HOSTNAME:`. The hostname can be preceded by a username and an `@` to access the remote tape drive as that user, if you have permission to do so (typically an entry in that user's `~/.rhosts` file).

-p, --pass-through
Run in copy-pass mode.

-r, --rename Interactively rename files.

-R [*user*][:.][*group*], --owner [*user*][:.][*group*]
In copy-out and copy-pass modes, set the ownership of all files created to the specified user and/or group. Either the user or the group, or both, must be present. If the group is omitted but the **:** or **.** separator is given, use the given user's login group. Only the super-user can change files' ownership.

-s, --swap-bytes
In copy-in mode, swap the bytes of each half-word (pair of bytes) in the files.

-S, --swap-halfwords
In copy-in mode, swap the half-words of each word (4 bytes) in the files.

-t, --list Print a table of contents of the input.

-u, --unconditional
Replace all files, without asking whether to replace existing newer files with older files.

-v, --verbose List the files processed, or with **-t**, give an `ls -l` style table of contents listing. In a verbose table of contents of a `ustar` archive, user and group names in the archive that do not exist on the local system are replaced by the names that correspond locally to the numeric `UID` and `GID` stored in the archive.

-V, --dot Print a **.** for each file processed.

--version Print the `cpio` program version number and exit.

The long-named options can be introduced with + as well as −−, for compatibility with previous releases. Eventually support for + will be removed, because it is incompatible with the POSIX.2 standard.

COPYRIGHT

NAME

`cron` – daemon to execute scheduled commands

SYNOPSIS

`cron`

DESCRIPTION

`cron` should be started from `/bin/rc`. It will return immediately, so you don't need to start it with '&'.

`cron` searches `/usr/cron/tabs` for `crontab` files which are named after accounts in `/etc/passwd`; crontabs found are loaded into memory. `cron` also searches for `/etc/crontab` which is in a different format (see `crontab(5)`). `cron` then wakes up every minute, examining all stored crontabs, checking each command to see if it should be run in the current minute. When executing commands, any output is mailed to the owner of the `crontab` (or to the user named in the `MAILTO` environment variable in the crontab, if such exists).

Additionally, `cron` checks each minute to see if its spool directory's `modtime` (or the `modtime` on `/etc/crontab`) has changed, and if it has, `cron` will then examine the `modtime` on all crontabs and reload those which have changed. Thus `cron` need not be restarted whenever a crontab file is modified. Note that the `crontab(1)` command updates the `modtime` of the spool directory whenever it changes a `crontab`.

SEE ALSO

Utility Programs - `cron`

File Formats - `crontab`

AUTHOR

Paul Vixie

NAME

crontab – maintain crontab files for individual users

SYNOPSIS

crontab [–u user] file

crontab [–u user] { –l | –r | –e }

DESCRIPTION

crontab is the program used to install, uninstall or list the tables used to drive the cron() daemon in Vixie Cron. Each user can have their own crontab, and though these are files in /usr/cron/tabs, they are not intended to be edited directly.

If the allow file exists, then you must be listed therein in order to be allowed to use this command. If the allow file does not exist but the deny file does exist, then you must not be listed in the deny file in order to use this command. If neither of these files exists, then depending on site-dependent configuration parameters, only the super user will be allowed to use this command, or all users will be able to use this command.

If the **–u** option is given, it specifies the name of the user whose crontab is to be tweaked. If this option is not given, crontab examines "your" crontab, i.e., the crontab of the person executing the command. Note that su (see su(1)) can confuse crontab and that if you are running inside a su session you should always use the **–u** option for safety's sake.

The first form of this command is used to install a new crontab from some named file or standard input if the pseudo-filename "-" is given.

The **–l** option causes the current crontab to be displayed on standard output.

The **–r** option causes the current crontab to be removed.

The **–e** option is used to edit the current crontab using the editor specified by the VISUAL or EDITOR environment variables. After you exit from the editor, the modified crontab will be installed automatically.

SEE ALSO

Utility Programs - `cron`

File Formats - `crontab`

FILES

`/usr/cron/allow`

`/usr/cron/deny`

STANDARDS

The `crontab` command conforms to IEEE Std1003.2-1992 (POSIX). This new command syntax differs from previous versions of Vixie Cron, as well as from the classic SVR3 syntax.

DIAGNOSTICS

A fairly informative usage message appears if you run it with a bad command line.

AUTHOR

Paul Vixie

NAME

`cs` – produce file checksum

SYNOPSIS

cs [–*value* **]** *file*

DESCRIPTION

`cs` sums every byte of the named *file*, modulo 256. If no *value* is given, the result is printed to standard output. If a *value* is given, `cs` compares it to the actual `checksum` and exits with status 1 if the values are different, or with status 0 if the values are the same.

NAME

`csh` – a shell (command interpreter) with C-like syntax

SYNOPSIS

tcsh [-bcefinstvVxX] [*arg* ...]

DESCRIPTION

LynxOS supports an extended version of the `csh` shell called `tcsh`. `tcsh` provides additional facilities that are described in `tcsh`(1). However, you should refer to this `csh` manual page for information about the core functionality of `tcsh`.

`csh` is a command language interpreter incorporating a history mechanism (see History Substitutions), job control facilities (see Jobs), interactive file name and user name completion (see File Name Completion), and a C-like syntax. It is used both as an interactive login shell and a shell script command processor.

Argument list processing If the first argument (argument 0) to the shell is '–' then this is a login shell. The flag arguments are interpreted as follows:

OPTIONS

–b This flag forces a "`break`" from option processing, causing any further shell arguments to be treated as non-option arguments. The remaining arguments will not be interpreted as shell options. This may be used to pass options to a shell script without confusion or possible subterfuge. The shell will not run a set-user ID script without this option.

–c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in `argv`.

–e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file `.cshrc` in the invoker's home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A '\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the verbose variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the echo variable to be set, so that commands are echoed immediately before execution.

-V Causes the verbose variable to be set even before `.cshrc` is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments, if arguments remain but none of the **-c**, **-i**, **-s**, or **-t** options were given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '`$0`'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a '`standard`' shell if the first character of a script is not a '`#`', i.e. if the script does not start with a comment. Remaining arguments initialize the variable `argv`.

An instance of `csh` begins by executing commands from the file `/etc/csh.cshrc` and, if this is a login shell, `/etc/csh.login`. It then executes commands from `csh` in the home directory of the invoker, and, if this is a login shell, the file `.login` in the same location. It is typical for users on dumb terminals to put the command "`stty crt`" in their `.login` file, and to also invoke `tset`(1) there.

In the normal case, the shell will begin reading commands from the terminal, prompting with '`% `'. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the files `.logout` in the user's home directory and `/etc/csh.logout`.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '`&`' '`|`' '`;`' '<' '>' '`(`' '`)`' form separate words. If doubled in '`&&`', '`||`', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with '\'. A newline preceded by a '\' is equivalent to a space character.

Strings enclosed in matched pairs of quotations, '`'`', '`` ` ``' or '`"`', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of '`` ` ``' or '`"`' characters a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '`#`' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by '\' and in quotations using '`` ` ``', '`'`', and '`"`'.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by '`|`' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ';', and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an '&'.

Any of the above may be enclosed within parentheses '()' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with '||' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See Expressions.)

Jobs

The shell associates a job with each pipeline. It keeps a table of current jobs, printed by the jobs command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may hit the key **Control-z** which sends a `STOP` signal to the current job. The shell will then normally indicate that the job has been '`Stopped`', and print another prompt. You can then manipulate the state of this job, putting it in the background with the `bg` command, or run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`. A **Control-z** takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is another special key **Control-y** which does not generate a `STOP` signal until a program attempts to read it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "`stty tostop`". If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character '%' introduces a job name. If you wish to refer to job number 1, you can name it as '`%1`'. Just naming a job brings it to the foreground; thus '`%1`' is a synonym for '`fg %1`', bringing job 1 back into the foreground. Similarly saying '`%1 &`' resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus '`%ex`' would normally restart a suspended `ex`(1) job, if there were only one suspended job whose name began with the string '`ex`'. It is also possible to say '`%?string`' which specifies a job whose text contains string, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a '+' and the previous job with a '–'. The abbreviation '`%+`' refers to the current job and '`%-`' refers to the previous job. For close analogy with the syntax of the history mechanism (described below), '`%%`' is also a synonym for the current job.

The job control mechanism requires that the `stty`(1) option new be set. It is an artifact from a new implementation of the tty driver which allows generation of interrupt characters from the keyboard to tell jobs to stop. See `stty`(1) for details on setting options in the new tty driver.

Status reporting

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable `notify`, the shell will notify you immediately of changes of status in background jobs. There is also a shell command `notify` which marks a single process so that its status changes will be immediately reported. By default notify marks the current process; simply say '`notify`' after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that '`You have stopped jobs.`' You may use the jobs command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

File Name Completion

When the file name completion feature is enabled by setting the shell variable `filec` (see `set`), `csh` will interactively complete file names and user names from unique prefixes, when they are input from the terminal followed by the escape character (the **Esc** key, or **Control-[**) For example, if the current directory looks like

```
DSC.OLD  bin      cmd      lib      xmpl.c DSC.NEW  chaosnet
cmtest   mail     xmpl.o   bench    class  dev      mbox
xmpl.out
```

and the input is

```
% vi ch<escape>
```

`csh` will complete the prefix “`ch`” to the only matching file name “`chaosnet`”, changing the input line to

```
% vi chaosnet
```

However, given

```
% vi D<escape>
```

`csh` will only expand the input to

```
% vi DSC.
```

and will sound the terminal bell to indicate that the expansion is incomplete, since there are two file names matching the prefix “`D`”.

If a partial file name is followed by the end-of-file character (usually **Control-d**), then, instead of completing the name, `csh` will list all file names matching the prefix. For example, the input

```
% vi D<control-D>
```

causes all files beginning with “`D`” to be listed:

```
DSC.NEW   DSC.OLD
```

while the input line remains unchanged.

The same system of escape and end-of-file can also be used to expand partial user names, if the word to be completed (or listed) begins with the character "~". For example, typing

```
cd ~ro<escape>
```

may produce the expansion

```
cd ~root
```

The use of the terminal bell to signal errors or multiple matches can be inhibited by setting the variable `nobeep`.

Normally, all files in the particular directory are candidates for name completion. Files with certain suffixes can be excluded from consideration by setting the variable `fignore` to the list of suffixes to be ignored. Thus, if `fignore` is set by the command

```
% set fignore = (.o .out)
```

then typing

```
% vi x<escape>
```

would result in the completion to

```
% vi xmpl.c
```

ignoring the files `xmpl.o` and `xmpl.out`. However, if the only completion possible requires not ignoring these suffixes, then they are not ignored. In addition, `fignore` does not affect the listing of file names by **Control-d**. All files are listed regardless of their suffixes.

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character '`!`' and may begin anywhere in the input stream (with the proviso that they do not nest.) This '`!`' may be preceded by an '\' to prevent its special meaning; for convenience, a '`!`' is passed unchanged when it is followed by a blank, tab, newline, '=' or '`(`'. (History substitutions also occur when an input line begins with ↑. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of which is controlled by the `history` variable; the previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

```
9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an '`!`' in the prompt string.

With the current event 13 we can refer to previous events by event number '`!11`', relatively as in '`!-2`' (referring to the same event), by a prefix of a command word as in '`!d`' for event 12 or '`!wri`' for event 9, or by a string contained in a word in the command as in '`!?mic?`' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '`!!`' refers to the previous command; thus '`!!`' alone is essentially a redo.

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*th argument
↑	first argument
$	last argument
%	word matched by (immediately preceding) ?*searchstring*? search.
x – y	range of words
–*y*	abbreviates the range of words from 0 to *y*.
*	abbreviates ↑–$, or nothing if only 1 word in event
*x**	abbreviates *x*–$
x–	like *x** but omitting the last argument

The colon (:) separating the event specification from the word designator can be omitted if the argument selector begins with a ↑, $, *, – or %. After the optional word designator can be placed a sequence of modifiers, each preceded by a colon (:). The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
e	Remove all but the extension '.xxx' part.
s/*l*/*r*/	Substitute *l* for *r*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. 'g&'.

`p`	Print the new command line but do not execute it.
`q`	Quote the substituted words, preventing further substitutions.
`x`	Like `q`, but break into words at blanks, tabs and newlines.

Unless preceded by a '`g`' the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of '`/`'; a '`\`' quotes the delimiter into the *l* and *r* strings. The character '`&`' in the right hand side is replaced by the text from the left. A '`\`' quotes '`&`' also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in '`!?`*s*'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing '`?`' in a contextual scan.

A history reference may be given without an event specification, e.g. '`!$`'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus

```
!?foo?↑ !$
```

gives the first and last arguments from the command matching `?foo?`.

A special abbreviation of a history reference occurs when the first nonblank character of an input line is a ↑. This is equivalent to `!:`*s*↑, after which you enter a search string followed by ↑ and a replacement string. This provides a convenient shorthand for substitutions on the text of the previous line. Thus, ↑`lb`↑`lib` fixes the spelling of `lib` in the last-executed command. Finally, a history substitution may be surrounded with `{` and `}` if you wish to insulate it from the characters that follow. Thus, after the command

```
ls -ld ~paul
```

we might enter

!{l}a

to obtain the same result as entering

ls -ld ~paula

Note that entering `!la` would have looked for a command starting with `la`.

Quotations with ' and "

The quotation of strings by '`'`' and '`"`' can be used to prevent all or some of the remaining substitutions. Strings enclosed in '`'`' are protected from any further interpretation. Strings enclosed in '`"`' may be expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitition below) does a '`"`' quoted string yield parts of more than one word; '`'`' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the `alias` and `unalias` commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for '`ls`' is '`ls -l`' the command '`ls /usr`' would map to '`ls -l /usr`', the argument list here being undisturbed. Similarly if the alias `lookup` was defined with

alias lookup 'grep \!* /etc/passwd'

then

lookup bill

would map to

```
grep bill /etc/passwd
```

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can '`alias print 'pr \!* | lpr'`' to make a command which runs `pr` for its file arguments, sending the output to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the `argv` variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the `set` and `unset` commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the `verbose` variable is a toggle which causes command input to be echoed. The setting of this variable results from the **-v** command line option.

Other operations treat variables numerically. The '`@`' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '`$`' characters. This expansion can be prevented by preceding the '`$`' with a '\' except within a pair of '`"`' (double quotation characters) where it always occurs, and within '`'`'s where it never occurs. Strings quoted by '`` ` ``' are interpreted later (see

Command substitution below) so '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in '"' or given the ':q' modifier the results of variable substitution may eventually be command and filename substituted. Within '"', a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the ':q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$*name*
${*name*}
: Are replaced by the words of the value of variable name, each separated by a blank. Braces insulate name from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter. If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$*name*[*selector*]
${*name*[*selector*]}
: May be used to select only some of the words from the value of *name*. The selector is subjected to '$' substitution and may consist of a single number or two numbers separated by a '–'. The first word of a variables value is numbered '1'. If the first number of a range is omitted it defaults to '1'. If the second number of a range

is omitted it defaults to the numeric value corresponding to the last word in *name* (`$#name`). The selector '*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

`$#`*name*
`${#`*name*`}` Gives the number of words in the variable. This is useful for later use in a '`$argv[`*selector*`]`'.

`$0` Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

`$`*number*
`${`*number*`}` Equivalent to '`$argv[`*number*`]`'.

`$*` Equivalent to '`$argv[*]`'. The modifiers '`:e`', '`:h`', '`:t`', '`:r`', '`:q`' and '`:x`' may be applied to the substitutions above as may '`:gh`', '`:gt`' and '`:gr`'. If braces '`{}`' appear in the command form then the modifiers must appear within the braces. The current implementation allows only one '`:`' modifier on each '`$`' expansion.

The following substitutions may not be modified with '`:`' modifiers.

`$?`*name*
`${?`*name*`}` Substitutes the string '`1`' if name is set, '`0`' if it is not.

`$?0` Substitutes '`1`' if the current input filename is known, '0' if it is not. Substitute the (decimal) process number of the (parent) shell.

`$<` Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in '`'. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within double-quote characters (`""`), only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters '`*`', '`?`', '`[`' or '`{`' or begins with the character '`~`', then that word is a candidate for filename substitution, also known as '`globbing`'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters '`*`', '`?`' and '`[`' imply pattern matching, the characters '`~`' and '`{`' being more akin to abbreviations.

In matching filenames, the character '`.`' at the beginning of a filename or immediately following a '`/`', as well as the character '`/`' must be matched explicitly. The character '`*`' matches any string of characters, including the null string. The character '`?`' matches any single character. The sequence '`[...]`' matches any one of the characters enclosed. Within '`[...]`', a pair of characters separated by '**–**' matches any character lexically between the two.

The character '`~`' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '`~`' it expands to the invokers home directory as reflected in the value of the variable `home`. When followed by a name consisting of letters, digits and '**–**' characters the shell searches for a user with that name and substitutes their home directory; thus '`~ken`' might expand to ‘`/usr/ken`’ and ‘`~ken/chmach`’ to ‘`/usr/ken/chmach`’. If the character ‘`~`’ is followed by a character other than a letter or ‘`/`’ or appears not at the beginning of a word, it is left undisturbed.

The metanotation ‘`a{b,c,d}e`’ is a shorthand for ‘`abe ace ade`’. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus ‘`~source/s1/{oldls,ls}.c`’ expands to

```
/usr/source/s1/oldls.c   /usr/source/s1/ls.c
```

whether or not these files exist without any chance of error if the home directory for ‘`source`’ is ‘`/usr/source`’. Similarly ‘`../{memo,*box}`’ might expand to ‘`../memo ../box ../mbox`’. (Note that ‘`memo`’ was not sorted with the results of matching ‘`*box`’.) As a special case ‘`{`’, ‘`}`’ and ‘`{}`’ are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

<name — Open file name (which is first variable, command and filename expanded) as the standard input.

<<word — Read the shell input up to a line which is identical to word. Word is not subjected to variable, filename or command substitution, and each input line is compared to word before any substitutions are done on this input line. Unless a quoting ‘\’, ‘"’, ‘ or ‘`’ appears in word variable and command substitution is performed on the intervening lines, allowing ‘\’ to quote ‘$’, ‘\’ and ‘`’. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

`>`*name*
`>!`*name*
`>&`*name*
`>&!`*name*
: The file name is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

  If the variable `noclobber` is set, then the file must not exist or be a character special file (e.g. a terminal or ‘`/dev/null`’) or an error results. This helps prevent accidental destruction of files. In this case the ‘`!`’ forms can be used and suppress this check.

  The forms involving ‘`&`’ route the diagnostic output into the specified file as well as the standard output. Name is expanded in the same way as ‘<’ input filenames are.

`>>`*name*
`>>&`*name*
`>>!`*name*
`>>&!`*name*
: Uses file name as standard output like ‘>’ but places output at the end of the file. If the variable `noclobber` is set, then it is an error for the file not to exist unless one of the ‘`!`’ forms is given. Otherwise similar to ‘>’.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The ‘<<’ mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. Note that the default standard input for a command run detached is not modified to be the empty file `/dev/null`; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see Jobs above).

Diagnostic output may be directed through a pipe with the standard output. Simply use the form ‘`|&`’ rather than just ‘`|`’.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the `@`, `exit`, `if`, and `while` commands. The following operators are available:

|| && |↑ & == != =~ !~ <= >= < > << >> + * / % ! ~ ( )

Here the precedence increases to the right, '==' '!=' '=~' and '!~', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '–', '*' '/' and '%' being, in groups, at the same level. The '==' '!=' '=~' and '!~' operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==' except that the right hand side is a pattern (containing, e.g. '*' or '?' wildcard characters and character classes in '[...]' against which the left hand operand is matched. This reduces the need for use of the switch statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in '{' and '}' and file enquiries of the form

–*l name*

where *l* is one of:

`r`	read access
`w`	write access
`x`	execute access
`e`	existence
`o`	ownership

`z`	zero size
`f`	plain file
`d`	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. '`0`'. Command executions succeed, returning true, i.e. '`1`', if the command exits with status 0, otherwise they fail, returning false. If more detailed status information is required then the command should be executed outside of an expression, after which the `status` variable should be tested.

Control flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The `foreach`, `switch`, and `while` statements, as well as the `if-then-else` form of the `if` statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, any backward-flowing `goto` statements will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

`alias`
`alias` *name*
`alias` *name wordlist*

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified

	wordlist as the alias of name; wordlist is command and filename substituted. Name is not allowed to be `alias` or `unalias`.
`alloc`	Shows the amount of dynamic memory acquired, broken down into used and free memory. With an argument shows the number of free and used blocks in each size category. The categories start at size 8 and double at each step. This command's output may vary across system types, since systems other than the VAX may use a different memory allocator.
`bg` `bg %`*job...*	Puts the current or specified jobs into the background, continuing them if they were stopped.
`break`	Causes execution to resume after the end of the nearest enclosing `foreach` or `while` statement. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.
`breaksw`	Causes a break from a switch, resuming after the `endsw`.
`case` *label*:	A label in a switch statement as discussed below.

`cd`
`cd` *name*
`chdir`
`chdir` *name*
	Change the shell's working directory to directory *name*. If no argument is given then change to the home directory of the user. If *name* is not found as a subdirectory of the current directory (and does not begin with '`/`', '`./`' or '`../`'), then each component of the variable `cdpath` is checked to see if it has a subdirectory that goes by the given *name*. Finally, if all else fails but *name* is a shell variable whose value begins with '`/`', then this is tried to see if it is a directory.

`continue`	Continue execution of the nearest enclosing `while` or `foreach` statement. The rest of the commands on the current line are executed.

`default:`	Labels the default case in a `switch` statement. The `default` statement should come after all `case` labels.
`dirs`	Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.
`echo` *wordlist* `echo` **`-n`** *wordlist*	The specified words are written to the standard output, separated by spaces, and terminated with a newline unless the **`-n`** option is specified.

`else`
`end`
`endif`
`endsw`
	See the description of the `foreach`, `if`, `switch`, and `while` statements below.

`eval` *arg...*	(As in `sh`.) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See `tset`(1) for an example of using `eval`.
`exec` *command*	The specified command is executed in place of the current shell.
`exit` `exit (`*expr*`)`	The shell exits either with the value of the status variable (first form) or with the value of the specified *expr* (second form).
`fg` `fg %`*job...*	Brings the current or specified jobs into the foreground, continuing them if they were stopped.

`foreach` *name*`(`*wordlist*`)`
`...`
`end`
	The variable *name* is successively set to each member of *wordlist* and the sequence of commands between `foreach` and its matching `end` are executed. (Both `foreach` and `end` must appear alone on separate lines.) The builtin command `continue` may be used to

continue the loop prematurely and the builtin command `break` to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with '`?`' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

`glob` *wordlist*
: Like echo but no '\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

`goto` *word*
: The specified word is filename and command expanded to yield a string that mates to a label somewhere. The shell rewinds its input as much as possible, searching for a line of the form '*label*`:`' possibly preceded by space characters or tabs. Execution continues after the matched line.

`hashstat`
: Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding `exec` functions). An `exec` is attempted for each component of the path where the hash function indicates a possible hit, and in each component which does not begin with a '`/`'.

`history`
`history` *n*
`history -r` *n*
`history -h` *n*
: Displays the history event list; if *n* is given only the *n* most recent events are printed. The **-r** option reverses the order of printout to be most recent first rather than oldest first. The **-h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourcing using the **-h** option to `source`.

`if (`*expr command*`)`
: If the specified expression evaluates true, then the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the `if` command. *command* must be a simple command, not a pipeline, a command

list, or a parenthesized command list. Input/output redirection occurs even if *expr* is false, when *command* is not executed (this is a bug).

`if (`*expr*`) then`
`...`
`elseif (`*expr2*`)then`
`...`
`else`
`...`
`endif`

If the specified *expr* is true then the commands to the first else are executed; otherwise if *expr2* is true then the commands to the second else are executed, etc. Any number of `else-if` pairs are possible; only one `endif` is needed. The `else` part is likewise optional. (The words `else` and `endif` must appear at the beginning of input lines; the `if` must appear alone on its input line or after an `else`.)

`jobs`
`jobs -l`

Lists the active jobs; given the **-l** options, it lists process id's in addition to the normal information.

`kill %`*job*
`kill` *pid*
`kill` **-***sig pid...*
`kill -l`

Sends either the `TERM` (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in `/usr/include/signal.h`, stripped of the prefix "`SIG`"). The signal names are listed by "`kill -l`". There is no default, saying just '`kill`' does not send a signal to the current job. If the signal being sent is `TERM` (terminate) or `HUP` (hangup), then the job or process will be sent a `CONT` (continue) signal as well.

`limit`
`limit` *resource*
`limit` *resource maximum-use*
`limit -h`
`limit -h` *resource*
`limit -h` *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given. If the **-h** flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. Only the super-user may raise the hard limits, but a user may lower or raise the current limits within the legal range.

Possible values for *resource* currently are:

`cputime` the maximum number of cpu-seconds to be used by each process

`filesize` the largest single file which can be created

`datasize` the maximum growth of the data+stack region via `sbrk`(2) beyond the end of the program text

`stacksize`
the maximum size of the automatically extended stack region

`coredumpsize`
the size of the largest core dump that will be created

The possible values for *maximum-use* are a (floating point or integer) number followed by a scale factor. For all limits other than `cputime` the default scale is '`k`' or kilobytes (1024 bytes); a scale factor of '`m`' or megabytes may also be used. For `cputime`, the default scaling is '`seconds`', while '`m`' for minutes or '`h`' for hours, or a time of the form '`mm:ss`' giving minutes and seconds may be used.

For both resource names and scale factors, unambiguous prefixes of the names suffice.

`login` Terminate a login shell, replacing it with an instance of `/bin/login`. This is one way to log off, included for compatibility with `sh`(1).

`logout` Terminate a login shell. Especially useful if `ignoreeof` is set.

`nice`
`nice` + *number*
`nice` *command*
`nice` + *number command*

The first form sets the scheduling priority for this shell to 4. The second form sets the priority to the given *number*. The final two forms run *command* at priority 4 and number respectively. The greater the number, the less CPU attention the process will get. The super-user may accord a process more CPU time by specifying a negative priority by using '`nice` –*number* ...'. The supplied *command* is always executed in a sub-shell, and the restrictions placed on commands in simple `if` statements apply.

`nohup`
`nohup` *command* The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with '`&`' are effectively run this way so that they continue to run even when the parent shell is exited.

`notify`
`notify %`*job...* Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable notify is set.

`onintr`
`onintr -`
`onintr` *label*

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form '`onintr -`' causes all interrupts to be ignored. The final form causes the shell to execute a '`goto label`' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of `onintr` have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

`popd`
`popd +`*n*

Pops the directory stack, returning to the new top directory. With an argument '+*n*' discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

`pushd`
`pushd` *name*
`pushd` *n*

With no arguments, pushd exchanges the top two elements of the directory stack. Given a *name* argument, `pushd` changes to the new directory (ala `cd`) and pushes the old current working directory (as in `csw`) onto the directory stack. With a numeric argument, rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

`rehash`

Causes the internal hash table of the contents of the directories in the path variable to be recomputed. This is needed if new commands are added to directories in the path while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

`repeat` *count* `command`

The specified command which is subject to the same restrictions as the command in the one line if statement above, is executed count times. I/O redirections occur exactly once, even if count is 0.

`set`
`set` *name*
`set` *name*= *word*
`set` *name*[*index*]= *word*
`set` *name*=(*wordlist*)

The first form of the command shows the values of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single word. The fourth form sets the *index*th component of name to *word*; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single `set` command. Note however, that variable expansion happens for all arguments before any setting occurs.

`setenv`
`setenv` *name value*
`setenv` *name*

The first form lists all current environment variables. The last form sets the value of environment variable *name* to be value, a single string. The second form sets *name* to an empty string. The most commonly used environment variables `USER`, `TERM`, and `PATH` are automatically imported to and exported from the `csh` variables `user`, `term`, and `path`; there is no need to use `setenv` for these.

`shift`
`shift` *variable*

The members of `argv` are shifted to the left, discarding `argv[1]`. It is an error for `argv` not to be set or to have less than one word as value. The second form performs the same function on the specified *variable*.

`source` *name*
`source -h` *name*
The current shell (not a subshell) reads commands from the file *name*. (If a subshell were spawned, commands that set environmental variables could not affect the environment of the current shell in a lasting way.) Source commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a source at any level terminates all nested source commands. Source commands are not placed on the history list; the **-h** option causes the commands to be placed in the history list without being executed. For example, to impart to the environment of a shell the modifications made in a `.login` or `.cshrc` file, specify the modified file as the argument to a `source` command.

`stop`
`stop %`*job...*
Stops the current or specified *job* which is executing in the background.

`suspend`
Causes the shell to stop in its tracks, much as if it had been sent a stop signal with**Control-Z**. This is most often used to stop shells started by `su`(1).

```
switch ( string)
case strlabel:
...
breaksw
...
default:
...
breaksw
endsw
```

Each `case` label is successively matched, against the specified *string* which is first command and filename expanded. The file metacharacters ‘`*`’, ‘`?`’ and ‘[...]’ may be used in the case labels, which are variable expanded. If none of the labels match before a ‘`default`’ label is found, then the execution begins after the `default` label. Each case label and the default label must appear at the beginning of a line. The command `breaksw` causes execution to continue after the `endsw`. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the `endsw`.

`time`
`time` *command* — With no argument, a summary of time used by this shell and its children is printed. If arguments are given, the specified simple *command* is timed and a time summary as described under the `time` variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

`umask`
`umask` *value* — The file creation mask is displayed (first form) or set to the specified *value* (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

`unalias` *pattern* — All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by '`unalias *`'. It is not an error for nothing to be unaliased.

`unhash` — Use of the internal hash table to speed location of executed programs is disabled.

`unlimit`
`unlimit` *resource*
`unlimit -h`
`unlimit -h` *resource*
Removes any limitation upon *resource*. If no resource is specified, then all resource limitations are removed. If **-h** is given, the corresponding hard limits are removed. Only the super-user may use the **-h** option.

`unset` *pattern* — All variables whose names match the specified *pattern* are removed. Thus all variables are removed by '`unset *`'; this has noticeably distasteful side-effects. It is not an error for nothing to be unset.

`unsetenv` *pattern*
Removes all variables whose name match the specified *pattern* from the environment. See also the `setenv` command above and `printenv`(1).

`wait` All background jobs are waited for. It the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

`while (`*expr*`)`
`...`
`end` While the specified expression evaluates non-zero, the commands between the `while` and the matching `end` are evaluated. `break` and `continue` may be used to terminate or continue the loop prematurely. (The `while` and `end` must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the `foreach` statement if the input is a terminal.

`%`*job* Brings the specified job into the foreground.

`%`*job* `&` Continues the specified job in the background.

`@`
`@`*name*=*expr*
`@`*name*`[`*index*`]=`*expr*

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains '<', '>', '&' or '|' then at least this part of the expression must be placed within parentheses. The third form assigns the value of *expr* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

The operators '*=', '+=', etc are available as in C. The space separating the *name* from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix '++' and '--' operators increment and decrement *name* respectively, i.e. '`@ i++`'.

Pre-defined and environment variables

The following variables have special meaning to the shell. Of these, `argv`, `cwd`, `home`, `path`, `prompt`, `shell` and `status` are always set by the shell. Except for `cwd` and `status` this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

This shell copies the environment variable `USER` into the variable `user`, `TERM` into `term`, and `HOME` into `home`, and copies these back into the environment whenever the normal shell variables are reset. The environment variable `PATH` is likewise handled; it is not necessary to worry about its setting other than in the file `.cshrc` as inferior `csh` processes will import the definition of path from the environment, and re-export it if you then change it.

`argv` — Is set to the arguments to the shell upon its invocation, it is from this variable that positional parameters are substituted, i.e. '`$1`' is replaced by '`$argv[1]`', etc.

`cdpath` — Gives a list of alternate directories searched to find subdirectories in `chdir` commands.

`cwd` — The full pathname of the current directory.

`echo` — Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

`filec` — Enable file name completion.

`histchars` — Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character '`!`'. The second character of its value replaces the character ↑ in quick substitutions.

`history`	Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of history may run the shell out of memory. The last executed command is always saved on the history list.
`home`	The home directory of the invoker, initialized from the environment. The filename expansion of '~' refers to this variable.
`ignoreeof`	If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by **Control-d** keypresses.
`mail`	The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says '`You have new mail.`' if the file exists with an access time not greater than its modify time. If the first word of the value of mail is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes. If multiple mail files are specified, then the shell says '`New mail in name`' when there is mail in the file name.
`noclobber`	As described in the section Input/Output, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.
`noglob`	If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.
`nonomatch`	If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. '`echo [`' still gives an error.
`notify`	If set, the shell notifies asynchronously of job completions. The default behavior is to offer notices of job completion just before printing a new prompt.

`path`	Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no path variable then only full path names will execute. The usual search path is '`.`', '`/bin`' and '`/usr/bin`', but this may vary from system to system. For the super-user the default search path is '`/etc`', '`/bin`' and '`/usr/bin`'. A shell which is given neither the **`-c`** nor the **`-t`** option will normally hash the contents of the directories in the `path` variable after reading `.cshrc`, and each time the `path` variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to do a `rehash` or the commands may not be found.
`prompt`	The string which is printed before each command is read from an interactive terminal input. If a '`!`' appears in the string it will be replaced by the current event number unless a preceding '\' is given. Default is '`% `', or '`# `' for the superuser.
`savehist`	Is given a numeric value to control the number of entries of the history list that are saved in `~/.history` when the user logs out. Any command which has been referenced in this many events will be saved. During start up the shell sources `~/.history` into the history list enabling history to be saved across logins. Excessively large values of `savehist` will slow down the shell during start up.
`shell`	The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of Non-builtin Command Execution below.) Initialized to the (system-dependent) home of the shell.
`status`	The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `1`, all other builtin commands set `status` to `0`.

`time`	Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.
`verbose`	Set by the **–v** command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via `execve`(2). Each word in the variable `path` names a directory from which the shell will attempt to execute the command. If it is given neither a **–c** nor a **–t** option, the shell will hash the names in these directories into an internal table so that it will only try an `exec` in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via `unhash`), or if the shell was given a **–c** or **–t** option, and in any case for each directory component of `path` which does not begin with a '/', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus

```
(cd; pwd); pwd
```

prints the home directory; leaving you where you were (printing this after the home directory), while

```
cd; pwd
```

leaves you in your home directory. Parenthesized commands are most often used to prevent `chdir` from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands and a new shell is spawned to read it.

If there is an alias for shell then the words of the alias will be prepended to the argument list to form the shell command. The first word of the alias should be the full path name of the shell (e.g. '`$shell`'). Note that this is a special, late occurring, case of alias substitution, and only allows words to be prepended to the argument list without modification.

Signal handling

The shell normally ignores quit signals. Jobs running detached (either by `&` or the `bg` or `%... &` commands) are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. The shell's handling of interrupts and terminate signals in shell scripts can be controlled by `onintr`. Login shells catch the terminate signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file `.logout`.

AUTHOR

William Joy. Job control and directory stack features first implemented by J.E. Kulp of IIASA, Laxenburg, Austria, with different syntax than that used now. File name completion code written by Ken Greer, HP Labs. Eight-bit implementation Christos S. Zoulas, Cornell University.

FILES

`~/.cshrc`	Read at beginning of execution by each shell.
`~/.login`	Read by login shell, after '`.cshrc`' at login.
`~/.logout`	Read by login shell, at logout.
`/bin/sh`	Standard shell, for shell scripts not starting with a '`#`'.
`/tmp/sh*`	Temporary file for '<<'.

/etc/passwd	Source of home directories for '~name'.

LIMITATIONS

Word lengths — Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command which involves filename expansion is limited to one-sixth the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of alias substitutions on a single line to 20.

SEE ALSO

Utility Programs -	sh, tcsh
System Calls -	access(), execve(), fork(), killpg(), pipe(), sigvec(), umask(), getrlimit(), wait()
Device Files -	tty
File Formats -	a.out

HISTORY

csh appeared in 3BSD. It was a first implementation of a command language interpreter incorporating a history mechanism (see History Substitutions), job control facilities (see Jobs), interactive file name and user name completion (see File Name Completion), and a C-like syntax. There are now many shells which also have these mechanisms, plus a few more (and maybe some bugs too), which are available through the usenet, or with BSD as contributed software like the ksh (Korn shell).

BUGS

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (i.e. wrong) as the job may have changed directories internally.

Shell builtin functions are not stoppable and restartable. Command sequences of the form '`a ; b ; c`' are also not handled gracefully when stopping is attempted. If you suspend '`b`', the shell will then immediately executes '`c`'. This is especially noticeable if this expansion results from an alias. It suffices to place the sequence of parenthesized command blocks to force it to a subshell, i.e. '`( a ; b ; c )`'.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by '`?`', are not placed in the history list. Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '`|`', and to be used with '`&`' and '`;`' metasyntax.

It should be possible to use the '`:`' modifiers on the output of command substitutions. All and more than one '`:`' modifier should be allowed on '`$`' substitutions.

The way the `filec` facility is implemented is ugly and expensive.

NAME

`ctags` – generate editor tag file

SYNOPSIS

`ctags [ -BFwx]` *file ...*

DESCRIPTION

`ctags` locates all function definitions in the given C source code files and produces a table used by `vi` to quickly locate those functions during an editing session. Each line of the table is of the form:

function-name file-name search-pattern

The table is (always) written to the file tags in the current working directory.

OPTIONS

`-B` Generate backward search commands (`?pattern?`) in the tag table.

`-F` Generate forward search commands (`/pattern/`) in the tag table (default).

`-w` Suppress warning messages about duplicate tags. `ctags` ignores all but the first instance of a function name definition. This is most often a problem when a large program has `static` functions with identical names in different modules.

`-x` Causes an index file to be written to standard output. Each index file line is of the form

function-name file-name line-number

and can be used for a readable index to accompany a printed source listing. Note that the normal tag file is still generated when this option is chosen.

FILES

`tags`

SEE ALSO

Utility Programs - `vi`

NAME

`cut` – cut out selected fields of each line of a file

SYNOPSIS

cut -c *list* **[** *file ...* **]**

cut -f *list* **[-d** *char***]** **[-s]** **[** *file ...* **]**

DESCRIPTION

`cut` extracts columns or fields from each line of a file. Fixed-length columns are extracted using the **-c** option. Variable length fields are extracted using the **-f** option. The default field delimiter character is the tab character, and can be changed using the **-d** option. Cut reads standard input if no *files* are specified. In addition, a filename of "**-**" explicitly refers to standard input.

The list following the **-c** and **-f** options is a comma-separated list of integer field numbers, with an optional "**-**" to indicate ranges [e.g., **1,4,7**; **1-3,8**; **-5,10** (short for **1-5,10**); or **3-** (meaning third through last field)].

OPTIONS

-c *list* The *list* specifies character positions (e.g., -c1-72 would pass the first 72 characters of each line).

-f *list* The *list* is a list of fields assumed to be separated in the file by a delimiter character (see **-d**); e.g., **-f1**,7 copies the first and seventh field only. Lines with no field delimiters will be passed through intact unless the **-s** option is specified.

-d *char* Use *char* as the field delimiter (**-f** option only) instead of tab. Characters with special meaning to the shell must be quoted.

-s Suppresses lines with no delimiter characters in case of the **-f** option.

Either the **-c** or **-f** option must be specified.

EXAMPLES

To get a mapping of user IDs to names from the password file:

```
cut -d: -f1,5 /etc/passwd
```

To print the sizes of all files in the current directory:

```
ls -l | cut -c24-32,46-
```

DIAGNOSTICS

All error messages contain limits for line and field lengths.

SEE ALSO

Utility Programs - `grep, paste`

Library Functions - `getopt()`

NAME

`date` – display or set current date and time

SYNOPSIS

date [-u] [-d *n***] [-z** *m***] [** *newdate***] [+***format***]**

DESCRIPTION

`date` can be used to either set the current system date and time or to display it. If no arguments are given, then `date` will display only the current date and time.

A new date and time can be specified using any of the following:

```
ccyymmddhhmm[.ss]
yymmddhhmm[.ss]
mmddhhmm[.ss]
ddhhmm[.ss]
hhmm[.ss]
```

where `cc` is the number of the century (19 or 20); `yy` is the last two digits of the year; `mm` the number of the month (1-12); `dd` the day number in the month; `hh` the hour (0-23); `mm` the minute; and, optionally, `.ss` is the second. Wherever the year (`yy`), month (`mm`), day (`dd`), and/or second (`ss`) are omitted, the current values are used. If the century (`cc`) is omitted, it is assumed to be 19.

OPTIONS

-u Displays the current date in Greenwich Mean Time (GMT). When setting the current time, this flag prevents local transformations of the given time value.

-d *n* Sets the daylight saving method to *n*, which must be one of the following values:

- `0` (no daylight savings)
- `1` (USA)
- `2` (Australia)

`3`	(East Europe)
`4`	(Central Europe)
`5`	(Western Europe)

−z *m* Sets the time zone to *m*, given as minutes west of Greenwich (values for North America are, thus, positive).

+*format* When the format is specified, each field descriptor shall be replaced in the standard output by its corresponding value. All other characters shall be copied to the output line without change. The output shall always be terminated by a newline character. Format descriptors are:

`%a`	Locale’s abbreviated weekday name.
`%A`	Locale’s full weekday name.
`%b`	Locale’s abbreviated month name.
`%B`	Locale’s full month name.
`%c`	Locale’s appropriate date and time representation.
`%d`	Day of month as a decimal number (01-31).
`%D`	Date in the format *mm/dd/yy*.
`%e`	Day of month as a decimal number (1-31 in a two -digit field with leading space fill).
`%h`	A synonym for `%b`
`%H`	Hour (24-hour clock) as a decimal number (00-23).
`%I`	Hour (12-hour clock) as a decimal number (01-12).
`%j`	Day of the year as a decimal number (001-366).
`%m`	Month as a decimal number (01-12).
`%M`	Minute as a decimal number (00-59).
`%n`	A newline character.
`%p`	Locale’s equivalent of either AM or PM.

`%r`	12-Hour clock time (01-12) using the AM/PM notation; in the POSIX locale, this shall be equivalent to `"%I:%M:%S %p"`.
`%S`	Seconds as a decimal number (00-61).
`%t`	A tab character
`%T`	24-Hour clock time in the format HH:MM:SS.
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
`%w`	Weekday as a decimal number(0-6) where Sunday is the first day.
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number (00-53).
`%x`	Locale's appropriate date representation.
`%X`	Locale's appropriate time representation.
`%y`	Year offset in current century as a decimal number (00-99).
`%Y`	Year with century as a decimal number.
`%Z`	Time zone name.
`%%`	A percent sign character.

EXAMPLES

Sets the date to December 8, 1992 12:30 AM GMT, and the daylight saving method to USA.

```
date −d1 −u 9212080030
```

Leaves the current time the same, but changes the time zone to Central Standard Time.

```
date −z360
```

Sets the date to June 7, 2005 10:20:30 AM.

```
date 200506071020.30
```

NAME

`dd` – convert and copy a file

SYNOPSIS

dd [*operand*... **]**

DESCRIPTION

The `dd` utility copies the specified input file to the specified output file 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.

OPTIONS

None.

OPERANDS

The following operands are available:

if=*file*	Input file name; standard input by default.
of=*file*	Output file name; standard output by default.
ibs=*n*	Input block size. *n* bytes (default is 512)
obs=*n*	Output block size. n bytes (default is 512)
bs=*n*	Set both input and output block size, superseding both **ibs** and **obs**.
cbs=*n*	Conversion buffer size.
skip=*n*	Skip *n* input blocks before starting copy.
seek=*n*	Seek *n* blocks from beginning of output file before copying.
count=*n*	Copy only *n* input blocks.

conv=*value*[,*value*...]...

ascii	Convert EBCDIC to ASCII.
ebcdic	Convert ASCII to EBCDIC.
ibm	Map of ASCII to EBCDIC, which maps an ASCII character to its EBCDIC equivalent.
lcase	Map US ASCII alphabet to lowercase.
ucase	Map US ASCII alphabetics to uppercase.
swab	Swap every pair of bytes.
noerror	Do not stop processing on an error.
sync	Pad every input block to ibs.
value,value	Several comma-seperated conversions.

The option **cbs** is used only if ascii or ebcdic conversion is specified. In the former case, cbs characters are placed into the conversion buffer, converted to ASCII, and trailing blank(s) trimmed and newline added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blank(s) added to make up an output block of size **cbs**.

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2, respectively. A pair of numbers may be seperated by **x** to indicate a product.

ASYNCHRONOUS EVENTS

For `SIGINT`, `dd` writes status information to standard error before exiting. Otherwise, it takes default action for all other signals.

EXIT STATUS

The `dd` utility exits with one of the following values:

0	All input files were copied normally.
>0	The utility aborted because errors occured during copying or accessing files or directories. An error will be written to standard error for nonexistent files or directories,identical source and target, or permissions which do not allow the user to access source or target.

NAME

`devices` – show installed devices

SYNOPSIS

`devices`

DESCRIPTION

`devices` produces a list of all installed devices. For each, `devices` gives the device number, type, driver ID, use count (number of open descriptors), start address of the device memory area, size of the device information area (dynamically loaded devices only), and name of the device. Even though the STREAMS devices are installed as special "character" type devices, `devices` shows them as type `stdev`.

If `(no dev)` appears in the place of the device memory area start address, then that device failed to correctly install. A device can fail to install for numerous reasons: the hardware may not be present; the driver install routine could crash; not enough of some resource causes install to return `SYSERR`; etc.

NOTE

If a character device fails to install, its corresponding block device must also fail to install. It is not possible to have a block device install when its character device fails to install.

SEE ALSO

Utility Programs - `devinstall, mknod`

NAME

`devinstall` – install physical device

SYNOPSIS

devinstall –[bc] [–e *n***] –d** *n device*

devinstall –u –[bc] *dev-ID*

devinstall –a

DESCRIPTION

`devinstall` loads or unloads the information block of a physical device into the operating system. The device driver must have already been installed, either as part of the operating system or with `drinstall`.

In the first form, `devinstall` is told whether the device is a block or character device, the driver ID of the device driver, and optionally a value for the "extra" parameter word. *device* is a pathname for a device information block.

In the second form, `devinstall` unloads the selected device. The type of the device, either block or character, must be specified.

In the third form, `devinstall` installs all devices described in the file `/etc/devices`. The driver ID numbers are read from the standard input as tuples, "driver-name driver-ID". This information normally is supplied by `drinstall`, as follows:

drinstall –a | devinstall –a

OPTIONS

–b	Device to be installed is a block device.
–c	Device to be installed is a character device. When installing or removing devices, the device type must be given.
–e *n*	Supply value for the "extra" word. This value is passed to the device driver when the device is installed. Default is zero.

-d *n*	Driver ID number.
-u	Unload device instead of loading device. In this case, the device ID number is given, not the device information file pathname.
-a	Install all devices described in `/etc/devices`.

SEE ALSO

Utility Programs -	`drinstall, mknod, mount, umount`
System Calls -	`bdv_install(), bdv_uninstall(), cdv_install(), cdv_uninstall()`

NAME

`df` – show free disk space

SYNOPSIS

`df [-i] [` *file ...* **`]`**

DESCRIPTION

`df` displays the total space and amounts of used and free space on selected file systems. `df` reports disk usage in multiples of the block size for that file system. If file names a device on which a file system exists, `df` gives information for that file system. If file refers to a regular file or directory, `df` gives information for the file system containing that file

If no file arguments are given, then `df` produces a report for each device listed in the kernel mount table.

OPTIONS

`-i` Give information on free and used inodes as well as free and used space.

SEE ALSO

Utility Programs - `du`

NAME

`diff` – intelligent file comparison

SYNOPSIS

diff [-abBefHilnNrstTw] [-c *n***] [-C** *n***] [-F** *regexp***] [-I** *regexp***]**
[-S *name***]** *dir1 dir2*

diff [-abBefHintTw] [-c *n***] [-C** *n***] [-F** *regexp***] [-I** *regexp***]**
file1 file2

DESCRIPTION

`diff` compares text files line-by-line and reports differences between them. Differences may be reported in different selectable ways, but in general represent the minimum number of changes necessary to cause one file to become the same as the other. Files that do not appear to be text files are compared byte by byte until a difference is found; only the fact that a difference exists is reported.

In a simple file-to-file comparison, either (not both) of the pathnames may be "**–**", instructing `diff` to read standard input. If only one of the two files named is a directory, `diff` compares the file to a file in the directory with the same base name (if one exists).

If both files are directories, all component files of each with like names are compared normally as above. Common subdirectories and names that appear in only one of the two directories are listed. `diff` can also be instructed to continue the comparison recursively down all common subdirectories.

The results of file comparisons are by default reported as a modification description indicating the necessary changes in terms of line numbers to change, followed by the lines themselves. The changes described, if applied, cause *file1* to become identical to *file2*. The modification descriptions are as follows:

`n1an3,n4`	Append after line *n1* in *file1*, lines `n3` through `n4` from *file2*
`n1,n2dn3`	Delete lines `n1` through `n2` from *file1*

`n1,n2cn3,n4` Change lines `n1` through `n2` in `file1` to lines `n3` through `n4` in *file2*. This operation is a combination of delete and append.

When `diff` prints the affected lines, those from *file1* are preceded by a “<“ character, and those from *file2* are preceded by a “>” character.

OPTIONS

The following options are only meaningful for directory `diff`’s:

–l The output of each comparison is piped through `pr`. Differences in the directory contents are reported after all individual `diff`’s.

–N If a file appears in one directory but not the other, pretends that the file is present but empty in the other directory.

–r Recursively examine subdirectories with matching names.

–s Includes a list of files that are identical. Ordinarily, file-to-file comparisons that “succeed” are silent.

–S *name* Starts at the given file *name* in the directories. Useful when resuming aborted `diff`’s.

The remaining options affect output format or the methods of comparison. Output format options **–c**, **–C**, **–e**, **–f**, and **–n** are mutually exclusive. Additionally, the **–F** option is only useful with context `diff`’s.

–a Forces all files to be treated as text files.

–b Differences in amount of whitespace are ignored (any non-empty string of whitespace is equivalent to any other), and whitespace at the ends of lines is completely ignored.

–B Differences involving numbers of blank lines are ignored.

–c *n* Instructs `diff` to report differences with *n* surrounding lines of context. If *n* is not given, 3 surrounding lines are shown. Each difference is preceded by a line of asterisks, followed by the file names and last modification dates. Lines removed from `file1` are shown preceded by a dash ("**–**"). Lines added to *file2* are shown preceded by a plus sign ("**+**"). Lines that fit between identical contextual regions but are different are shown preceded by an exclamation point ("**!**").

–C *n*	Similar to **–c** above, but also prints the most recent line containing an alphabetic character or an underscore in the first column. Useful when applied to C program source, because such lines are often function declaration lines.
–e	Produces a script for the UNIX editor `ed` to change *file1* into *file2*.
–f	Produces a script that looks somewhat like an `ed` script (not enough, however, to work in that capacity), but with changes listed in the actual order they are encountered.
–F *regexp*	For each set of changes in a context `diff`, shows the most recent line matching the specified regular expression *regexp*. Regular expressions may be given with the same syntax used with `egrep`.
–H	Applies a heuristic technique for finding small dense groups of changes.
–i	Ignores alphabetic case in comparisons.
–I *regexp*	Ignores differences including lines matching the specified regular expression *regexp*.
–n	Output format is in RCS format, similar to **–f** but with a count of affected lines added for each change.
–t	Expands tabs to spaces in the output.
–T	Separates leading indicator characters from text lines with a tab instead of a space, in order to better preserve the formatting of the files.
–w	Ignores all horizontal whitespace.

DIAGNOSTICS

If `diff` finds no differences according to options in effect, it exits with status 0. If differences are encountered, exit status is 1. If `diff` exits due to some file or system error, exit status is 2.

COPYRIGHT

SEE ALSO

Utility Programs - `cmp, cc, diff3, egrep, ed`

NAME

`diff3` – three-way differential merge comparison

SYNOPSIS

`diff [-exEX3] [-i ]` *file1 file2 file3*

DESCRIPTION

`diff3` compares three input files line by line, finds groups of lines that differ, and reports each group of differing lines. Its output is designed to make it easy to inspect two different sets of changes to the same file.

If `diff3` determines that any of the files are binary (non-text) files, it reports an error.

When no options are given, `diff3` reports ranges of differing lines with one of the following codes:

====	all three files differ
====*1*	*file1* is different
====*2*	*file2* is different
====*3*	*file3* is different

The code will be followed by one or more of the following line that describe the differences:

f : n1 a	The difference is that text is appended after line *n1* in file *f*, where *f* = 1, 2 or 3, which indicates *file1*, *file2* or *file3*.
`f:n1, n2 c`	The difference is that text in the range line *n1* to line *n2* changed in file *f* as indicated above. If *n1* = *n2*, the range may be abbreviated to *n1*.

When one of the options below are given, `diff3` generates output as `ed` scripts.

OPTIONS

The following options are meaningful to `diff3`. They may only be used one at a time.

-e Generate an `ed` script that incorporates all the changes from *file2* to *file3* into *file1*.

-E Like **-e**, except bracket lines from overlapping changes from *file1* and *file3*. With this option, an overlapping change looks like this:

<<<<<<< *file1*

`<lines from` *file1*>

=======

`<lines from` *file3*>

>>>>>>> *file3*

-x Like **-x**, except output only the overlapping changes.

-X Like **-E**, except output only the overlapping changes. In other words, like **-x**, except bracket changes as in **-E**.

-3 Like **-e**, except output only the nonoverlapping changes.

In addition, the following option may be used in combination with the above options.

-i Act as System V `diff3` and appended the `ed` command sequence **wq** onto the script that it is created.

COPYRIGHT NOTICE

LynxOS `diff3` is actually GNU `diff3`, and is copyrighted ©1988 by Richard M. Stallman and the Free Software Foundation, Inc. As such, Lynx Real-Time Systems will make available for nominal postage and materials cost complete source code for all parts of `diff`. Furthermore, LynxOS users are hereby permitted to copy and distribute `diff` at will under these same terms; that is, notice that source is available from Lynx Real-Time Systems must be given along with any `diff3` distribution, or else the source itself. If `diff3` in source form is to be used as a component of some other software system, the Free Software Foundation should be contacted for details:

Free Software Foundation, Inc.
675 Massachusetts Ave.
Cambridge, MA 02139

Note that GNU `diff3` has been modified in small ways by Lynx Real-Time Systems, Inc., and thus although source for GNU `diff3` may be available elsewhere it will be different from LynxOS `diff3`. Lynx Real-Time Systems assumes no responsibility for copies of `diff3` obtained elsewhere.

SEE ALSO

Utility Programs - `diff, ed`

NAME

`dlbsd` – attach/detach a STREAMS Ethernet interface

SYNOPSIS

dlbsd -a *dlpi_device bsd_interface*
dlbsd -d *dlpi_device*
dlbsd -s *dlpi_device*

DESCRIPTION

`dlbsd` is used to control the association between a Pseudo BSD network interface and a STREAMS-based Ethernet device which supports the DLPI specifications. Normally the DLPI Ethernet device driver provides an interface between the Ethernet and the streams. With the **-a** option, the `dlbsd` command can be used to attach the Pseudo BSD network interface *bsd_interface* to the DLPI Ethernet device *dlpi_device* for TCP/IP protocol support. After this "attachment" is done, the `ifconfig` utility can be used to configure the *bsd_interface* for TCP/IP networking, just like in the case of a regular BSD network interface.

The `dlbsd` utility controls the `IFF_RUNNING` flag for the *bsd_interface*. This flag gets set when the "attach" operation is done.

With the **-d** option, the `dlbsd` command can be used to detach the previously associated *bsd_interface* from the specified *dlpi_device*. After this "detachment", the `IFF_RUNNING` flag for the `bsd_interface` is reset. The `dlbsd` command can be reused with the **-a** option to set this flag again. There is no need to reuse the `ifconfig` command again.

With the **-s** option, the `dlbsd` command can be used to check if the *dlpi_device* is attached to a *bsd_interface* or not.

DIAGNOSTICS

Appropriate diagnostic messages are displayed in the following cases:

- The *dlpi_device* is not the same as the configured device for the *bsd_interface* device for attachment.
- The *dlpi_device* is already attached when **-a** option is used.

- The *dlpi_device* is already detached when **-d** option is used.

SEE ALSO

Utility Programs - `ifconfig`

Device Facilities - `hybrid_ethernet`, `dlpilan`, `bdlan`

NAME

`dlsh` – Lynx proprietary command interpreter

SYNOPSIS

`dlsh`

DESCRIPTION

`dlsh` is a user shell which can be used in place of Bourne, Korn, or C shells. `dlsh` is a full-featured command interpreter roughly equivalent in power to the Korn shell.

SEE ALSO

Utility Programs - `sh, csh, bash`

File Formats - `.dlshrc, .Login`

NAME

dosread, doswrite,
dosdir – manipulate DOS diskettes

SYNOPSIS

dosread [-a] *dev file ...*

doswrite [-a] *dev file*

dosdir [-lr] *dev* **[** *file ...* **]**

DESCRIPTION

The program dosread reads one or more *files* from a DOS- format diskette in drive *dev* and writes the contents to standard output. The file name may be given with LynxOS-style (slashes) or DOS-style (reverse slashes) directory separators. The file name cannot begin with a directory separator; DOS pathnames are always assumed to start from the root directory of the disk.

The aliases doswrite and dosdir allow other DOS manipulation functions. doswrite reads from standard input and writes to the specified DOS *file*. dosdir lists the contents of the specified *file*, which of course should be a DOS directory.

OPTIONS

-a (For dosread and doswrite only.) Enables conversion between LynxOS-style (line feed) and DOS-style (carriage return & line feed) line terminators.

-l (dosdir only.) Produces long directory listings instead of just file names.

-r (dosdir only.) Recursively descends a directory tree, listing all files found.

EXAMPLES

To list the root directory of a high density DOS diskette in drive 0 ("A" to DOS):

```
dosdir -l /dev/fd1200.0
```

To copy the file `somefile.xyz` from a 360K DOS diskette in drive 1 ("B" to DOS) to the directory `/usr/somebody`, converting line terminators:

```
dosread -a /dev/fd360.1 somefile.xyz > /
/usr/somebody/somefile.xyz
```

To write the file `/tmp/somefile.xyz` to a 360K DOS diskette in drive 1 ("B" to DOS) to the DOS directory `\somebody`, converting line terminators:

```
doswrite -a /dev/fd360.1 somebody\somefile.xyz < /
/tmp/somefile.xyz
```

NOTE

The DOS utilities are currently only available for LynxOS x86 systems. Only 360K and 1.2MB 5-1/4" diskettes, and 720K and 1.44MB 3-1/2" disks are recognized. Hard disk partitions are not recognized.

NAME

`drinstall` – install device drivers/STREAMS modules

SYNOPSIS

drinstall –a

drinstall –bcm *file*

drinstall –u *driverID*

DESCRIPTION

`drinstall` loads or unloads device drivers or STREAMS modules into the operating system. In the first form, all drivers listed in the file `/etc/drivers` are installed. The driver names and ID numbers are printed to the standard output, usually for use by `devinstall`. For the composite character/block driver case, the number printed has the block driver number in the upper short while the character driver number is in the lower short. In the second form, `drinstall` loads the driver from `file`. In the third form, `drinstall` unloads the driver whose ID number is *driverID*.

OPTIONS

–a Load all drivers described in `/etc/drivers`. `drinstall` prints pairs of driver names and driver ID numbers to the standard output, for use by `devinstall`:

```
drinstall –a | devinstall –a
```

–b Driver named is a composite character/block device driver.

–c Driver named is a character device driver.

–m Driver named is a STREAMS module.

–u Unload driver given by *driverID*.

SEE ALSO

Utility Programs - `devinstall`

System Calls - `dr_install()`, `dr_uninstall()`

File Formats - `drivers`

NAME

`drivers` – show system device drivers

SYNOPSIS

`drivers`

DESCRIPTION

`drivers` produces a list of all installed device drivers in the system. For each, `drivers` gives the driver ID, type, number of controlled major devices, the start and size of driver code (dynamically installed drivers only), and the driver name.

Even though the STREAMS devices are installed as special “character” type devices, `devices` shows them as `stdev`. The STREAMS modules are shown as type `stmod`.

SEE ALSO

Utility Programs - `drinstall`

NAME

`du` – show disk usage

SYNOPSIS

`du [-asqdt] [` *directory ...* **`]`**

DESCRIPTION

`du` reports the amount of space consumed by files in the subtree beneath each directory in 1024 (1K) byte units. If no *directory* is given, "`.`" is assumed. By default, a sub-total is given for each directory encountered in each subtree.

`du` does not traverse hard or symbolic links.

OPTIONS

`-a`	Give the size of each file encountered, instead of just directories.
`-s`	Only give the total subtree size; overrides **`-a`**.
`-q`	Prevents error messages from being displayed.
`-d`	Shows the depth of each directory or file.
`-t`	Gives a grand total, if more than one *directory* is given.

SEE ALSO

Utility Programs - `df, ls, shownode`

NAME

dump - dump selected parts of an object

SYNOPSIS

dump [-agfohsrltcL] [-d *number***] [+d** *number***] [-n** *name***] [-p] [-t** *index***] [+t** *index***] [-u] [-v] [-k] [-z** *name* **[[,]** *number***]] [+z** *number***]** *files*

DESCRIPTION

dump displays selected parts of the object files specified by *files*. *files* is a list of object, executable, and/or archive files. For archive files, each of its constituent object files will be dumped. dump can handle both a.out and coff.

OPTIONS

For each of the given object files, dump will dump selected parts according to the following options:

-a Dump the archive header of each object file in an archive file.

-g Dump the global symbols in the symbol table of an archive file.

-f Dump each file header.

-o Dump each optional header.

-h Dump section headers.

-s Dump section contents.

-r Dump relocation information.

-l Dump line number information.

-t Dump symbol table entries.

-z *name* Dump line number entries for the function called *name*.

-c Dump the string table.

-L Print the (interpreted) contents of the `.lib` sections.

The following options modify the behavior of the options listed above:

-d *number* Dump section *number*, or, if the **+d** option is also given, dump sections starting at section *number* and ending at the section specified by **+d**.

+d *number* Dump sections starting either at section 0, or, if the **-d** option is also given, starting at the section specified by **-d**, and ending at section *number*.

-n *name* Dump only information about *name*. For the **-h**, **-s**, **-r**, and **-l** options, *name* refers to a section *name* (e.g. `.text`). For the **-t** option, *name* refers to a symbol name.

-p Do not print headings in the output.

-t *index* Dump the symbol entry at *index*, or, if the **+t** option is also given, dump the symbol entries starting at *index* and ending at the index specified by **+t**.

+t *index* Dump symbol entries starting either at index 0, or, if the **-t** option is also given, starting at the index specified by **-t**, and ending at index, *index*.

-u Underline the names of files.

-v Dump information using symbols rather than numbers (e.g. `N_TEXT` instead of 0x4). This option has no effect on the **-s** or **-o** options.

-k Do not display the auxiliary entries in the symbol table of a `coff` object or executable.

-z *name,number*

For function *name*, dump line number entry *number*, or, if the **+z** option is also given, dump the line number entries starting at *number* and ending at the number specified by **+z**.

+z *number* If the **-z** option is not given, dump line number entries starting at 0 and ending at *number*. Otherwise, for the function name specified by the **-z** option, dump line number entries starting either at 0, or, if a number was specified with the **-z** option, starting at that number, and ending at *number*.

Without any options, `dump` will show the maximum amount of information about each object file.

SEE ALSO

Utility Programs – `od, nm, disasm, libr`

NAME

`dynaminst` – dynamically install drivers and devices

SYNOPSIS

dynaminst [-v] [*insttab* **]**

DESCRIPTION

`dynaminst` dynamically installs drivers and devices and creates special files (i.e. nodes) according to the installation file `insttab`. If `insttab` is not specified, then `dynaminst` reads the installation file `/etc/insttab`.

The installation file consists of entries that describe the device drivers to install, the devices to install under each driver, and the special files (i.e. nodes) to create for each device. If `dynaminst` is unable to dynamically install any of the listed drivers, it will print an error message on the standard error channel and terminate. If it is unable to dynamically install a device or to create a device's special file, it will print a warning message on the standard error channel and continue with the next entry in the installation file.

As a special case, if `dynaminst` is unable to install any devices under a particular driver, either because none were listed or because all devices listed for the driver failed installation, then `dynaminst` will dynamically un-install that driver and print an accompanying message on the standard error channel.

OPTIONS

-v Print verbose information during the installation.

NOTE

`dynaminst` provides an alternate but more automated mechanism for dynamically installing drivers and devices into the operating system. The utilities `devinstall` and `drinstall` may also be used for this purpose.

SEE ALSO

Utility Programs -	`devinstall, devices, drinstall, drivers, mknod`
System Calls -	`bdv_install(), bdv_uninstall(), cdv_install(), cdv_uninstall(), dr_install(), mknod()`
File Formats -	`insttab`

NAME

`eject` – eject the floppy disk from a SPARC system

SYNOPSIS

`eject`

DESCRIPTION

For LynxOS SPARC systems only.

`eject` is used to eject a floppy disk from the disk drive. `eject` issues an `ioctl` to the floppy disk driver, `/dev/fd0`, in order to cause the ejection to occur.

NOTE

This command is provided only for SPARC floppy disk drives with auto-eject features.

To manually eject a floppy, stick a paper clip in the small hole on the front of the drive and push.

NAME

`emacs` – GNU Emacs text editor

SYNOPSIS

`emacs` `[`*command-line switches*`]` `[`*files ...* `]`

DESCRIPTION

`GNU Emacs` is a new version of `Emacs,` written by the author of the original (PDP-10) `Emacs,` Richard Stallman. Its user functionality encompasses everything other `Emacs` editors do, and it is easily extensible since its editing commands are written in Lisp.

`Emacs` has an extensive interactive help facility, but the facility assumes that you know how to manipulate `Emacs` windows and buffers. `CTRL-h` (backspace or `CTRL-h`) enters the Help facility. Help Tutorial (`CTRL-h t`) requests an interactive tutorial which can teach beginners the fundamentals of `Emacs` in a few minutes. Help Apropos (`CTRL-h a`) helps you find a command given its functionality, Help Character (`CTRL-h c`) describes a given character's effect, and Help Function (`CTRL-h f`) describes a given Lisp function specified by name.

`Emacs's` Undo can undo several steps of modification to your buffers, so it is easy to recover from editing mistakes.

`GNU Emacs's` many special packages handle mail reading (RMail) and sending (Mail), outline editing (Outline), compiling (Compile), running subshells within `Emacs` windows (Shell), running a Lisp read-eval-print loop (Lisp-Interaction-Mode), and automated psychotherapy (Doctor).

There is an extensive reference manual, but users of other Emacses should have little trouble adapting even without a copy. Users new to `Emacs` will be able to use basic features fairly rapidly by studying the tutorial and using the self-documentation features.

Emacs OPTIONS

The following options are of general interest:

file Edit *file.*

`+`*number* Go to the line specified by *number* (do not insert a space between the `+` sign and the number).

`-q` Do not load an `init` file.

`-u` *user* Load *user*'s `init` file.

`-t` *file* Use specified *file* as the terminal instead of using `stdin/stdout`. This must be the first argument specified in the command line.

The following options are lisp-oriented (these options are processed in the order encountered):

`-f` *function* Execute the lisp function *function.*

`-l` *file* Load the lisp code in the file *file.*

The following options are useful when running `Emacs` as a batch editor:

`-batch` *commandfile*
Edit in batch mode using the commands found in *commandfile.* The editor will send messages to stdout. This option must be the first in the argument list.

`-kill` Exit `Emacs` while in batch mode.

USING Emacs WITH X

`Emacs` has been tailored to work well with the X Window System. If you run `Emacs` from under X Window System, it will create its own X Window System to display in. You will probably want to start the editor as a background process so that you can continue using your original window.

`Emacs` can be started with the following X switches:

`-rn` *name* Specifies the program name which should be used when looking up defaults in the user's X resources. This must be the first option specified in the command line.

`-wn` *name* Specifies the name which should be assigned to the `Emacs` window.

`-r` Display the `Emacs` window in reverse video.

`-i` Use the "kitchen sink" bitmap icon when iconifying the `Emacs` window.

`-font` *font*, `-fn` *font*
Set the `Emacs` window's font to that specified by *font.* You will find the various `X` fonts in the `/usr/lib/X11/fonts` directory. Note that `Emacs` will only accept fixed width fonts. Under the X11 Release 4 font-naming conventions, any font with the value `m` or `c` in the eleventh field of the font name is a fixed width font. Furthermore, fonts whose name are of the form *width*x*height* are generally fixed width, as is the font *fixed.* See `xlsfonts` for more information.

When you specify a font, be sure to put a space between the switch and the font name.

`-b` *pixels* Set the `Emacs` window's border width to the number of pixels specified by *pixels.* Defaults to one pixel on each side of the window.

`-ib` *pixels* Set the window's internal border width to the number of pixels specified by *pixels.* Defaults to one pixel of padding on each side of the window.

`-w` *geometry*, `-geometry` *geometry*
Set the `Emacs` window's width, height, and position as specified. The geometry specification is in the standard X format; see `X` for more information. The width and height are specified in characters; the default is 80 by 24.

`-fg` *color* On color displays, sets the color of the text.

See the file `/usr/lib/X11/rgb.txt` for a list of valid color names.

`-bg` *color* On color displays, sets the color of the window's background.

`-bd` *color* On color displays, sets the color of the window's border.

`-cr` *color* On color displays, sets the color of the window's text cursor.

`-ms` *color* On color displays, sets the color of the window's mouse cursor.

`-d` *displayname*, `-display` *displayname*
Create the `Emacs` window on the display specified by *displayname*. Must be the first option specified in the command line.

`-nw` Tells `Emacs` not to use its special interface to X. If you use this switch when invoking `Emacs` from an `xterm` window, display is done in that window. This must be the first option specified in the command line.

You can set `X` default values for your `Emacs` windows in your `.Xresources` file (see `xrdb`). Use the following format:

```
emacs.keyword:value
```

where *value* specifies the default value of *keyword*. `Emacs` lets you set default values for the following keywords:

`font (class Font)`
Sets the window's text font.

`reverseVideo (class ReverseVideo)`
If `reverseVideo's` value is set to on, the window will be displayed in reverse video.

`bitmapIcon (class BitmapIcon)`
If `bitmapIcon's` value is set to on, the window will iconify into the "kitchen sink."

`borderWidth (class BorderWidth)`
Sets the window's border width in pixels.

`internalBorder (class BorderWidth)`
Sets the window's internal border width in pixels.

`foreground (class Foreground)`
For color displays, sets the window's text color.

`background (class Background)`
For color displays, sets the window's background color.

`borderColor (class BorderColor)`
For color displays, sets the color of the window's border.

`cursorColor (class Foreground)`
For color displays, sets the color of the window's text cursor.

`pointerColor (class Foreground)`
For color displays, sets the color of the window's mouse cursor.

`geometry (class Geometry)`
Sets the geometry of the `Emacs` window (as described above).

`title (class Title)`
Sets the title of the `Emacs` window.

`iconName (class Title)`
Sets the icon name for the `Emacs` window icon.

If you try to set color values while using a black and white display, the window's characteristics will default as follows: the foreground color will be set to black, the background color will be set to white, the border color will be set to grey, and the text and mouse cursors will be set to black.

USING THE MOUSE

The mouse button bindings for the `Emacs` window under X11 are shown in the table "Mouse Button Bindings Under X11".

Table 1: Mouse Button Bindings Under X11

Mouse Button	Function
left	Set point
middle	Paste text
right	Cut text into X cut buffer
SHIFT-middle	Cut text into X cut buffer
SHIFT-right	Paste text
CTRL-middle	Cut text into X cut buffer and kill it

Table 1: Mouse Button Bindings Under X11

Mouse Button	Function
CTRL-right	Select this window, then split it into two windows. Same as typing `CTRL-x 2`
CTRL-SHIFT-left	X buffer menu--hold the buttons and keys down, wait for menu to appear, select buffer, and release. Move mouse out of menu and release to cancel
CTRL-SHIFT-middle	X help menu--pop up index card menu for Emacs help
CTRL-SHIFT-right	Select window with mouse, and delete all other windows. Same as typing `CTRL-x 1`

MANUALS

You can order printed copies of the GNU Emacs Manual for $20.00/copy postpaid from the Free Software Foundation, which develops GNU software. Their address is:

Free Software Foundation
675 Mass Ave.
Cambridge, MA 02139

See the file `etc/DISTRIB` in the Emacs distribution for full ordering information. Your local Emacs maintainer might also have copies available. As with all software and publications from FSF, everyone is permitted to make and distribute copies of the Emacs manual. The TeX source to the manual is also included in the Emacs source distribution.

FILES

`/usr/local/emacs/src`	C source files and object files
`/usr/local/emacs/lisp`	Lisp source files and compiled files that define most editing commands. Some are preloaded; others are autoloaded from this directory when used.

`/usr/local/emacs/man`	sources for the Emacs reference manual.
`/usr/local/emacs/etc`	various programs that are used with GNU Emacs, and some files of information.
`/usr/local/emacs/etc/DOC.*`	contains the documentation strings for the Lisp primitives and preloaded Lisp functions of GNU Emacs. They are stored here to reduce the size of Emacs proper.
`/usr/local/emacs/etc/DISTRIB`	discusses GNU Emacs distribution and contains an order form for all of the software and manuals available from the Free Software Foundation.
`/usr/local/emacs/etc/DIFF`	discusses GNU Emacs vs. Twenex Emacs;
`/usr/local/emacs/etc/CCADIFF`	discusses GNU Emacs vs. CCA Emacs;
`/usr/local/emacs/etc/GOSDIFF`	discusses GNU Emacs vs. Gosling Emacs.
`/usr/local/emacs/etc/SERVICE`	lists people offering various services to assist users of GNU Emacs, including education, troubleshooting, porting, and customization.

These files also have information useful to anyone wishing to write programs in the Emacs Lisp extension language, which is

	documented in the GNU Emacs Lisp Manual.
`/usr/local/emacs/info`	files for the Info documentation browser (a subsystem of Emacs) to refer to. Currently not much of Unix is documented here, but the complete text of the Emacs reference manual is included in a convenient tree structured form.
`/usr/local/emacs/lock`	holds lock files that are made for all files being modified in Emacs, to prevent simultaneous modification of one file by two users.
`/usr/local/emacs/cpp`	the GNU cpp, needed for building Emacs on certain versions of Unix where the standard cpp cannot handle long names for macros.
`/usr/local/emacs/shortnames`	facilities for translating long names to short names in C code, needed for building Emacs on certain versions of Unix where the C compiler cannot handle long names for functions or variables.
`/usr/lib/X11/rgb.txt`	list of valid X color names.

BUGS

There is a mailing list, `bug-gnu-emacs@prep.ai.mit.edu` on the Internet (`ucbvax!prep.ai.mit.edu!bug-gnu-emacs` on UUCPnet), for reporting Emacs bugs and fixes. But before reporting something as a bug, please try to be sure that it really is a bug, not a misunderstanding or a deliberate feature. We ask you to read the section “Reporting Emacs Bugs” near the end of the reference manual (or Info system) for hints on how and when to report bugs. Also, include the version number of the Emacs you are running in every bug report that you send in.

Do not expect a personal answer to a bug report. The purpose of reporting bugs is to get them fixed for everyone in the next release, if possible. For personal assistance, look in the `SERVICE` file (see above) for a list of people who offer it.

Please do not send anything but bug reports to this mailing list. Send requests to be added to mailing lists to the special list `info-gnu-emacs-request@prep.ai.mit.edu` (or the corresponding UUCP address). For more information about Emacs mailing lists, see the file `/usr/local/emacs/etc/MAILINGLISTS`. Bugs tend actually to be fixed if they can be isolated, so it is in your interest to report them in such a way that they can be easily reproduced.

Bugs that I know about are: shell will not work with programs running in Raw mode on some Unix versions.

UNRESTRICTIONS

`Emacs` is free; anyone may redistribute copies of `Emacs` to anyone under the terms stated in the `Emacs` General Public License, a copy of which accompanies each copy of `Emacs` and which also appears in the reference manual.

Copies of `Emacs` may sometimes be received packaged with distributions of Unix systems, but it is never included in the scope of any license covering those systems. Such inclusion violates the terms on which distribution is permitted. In fact, the primary purpose of the General Public License is to prohibit anyone from attaching any other restrictions to redistribution of `Emacs.`

Richard Stallman encourages you to improve and extend `Emacs,` and urges that you contribute your extensions to the GNU library. Eventually GNU (Gnu's Not Unix) will be a complete replacement for Berkeley Unix. Everyone will be able to use the GNU system for free.

COPYRIGHT

AUTHORS

`Emacs` was written by Richard Stallman and the Free Software Foundation. Joachim Martillo and Robert Krawitz added the X features.

SEE ALSO

Utility Programs - `X, xlsfonts, xterm, xrdb`

NAME

`emit` – emit characters given their ASCII values.

SYNOPSIS

`emit` *value ...*

DESCRIPTION

`emit` writes the characters corresponding to the given numeric arguments on the standard output. *values* may be given in decimal, octal, or hexadecimal, using C syntax (leading `0` for octal, leading `0x` for hex). `emit` is generally used to send sequences of control characters to a device such as a printer.

NAME

`expand, unexpand` – add or remove tabs from files

SYNOPSIS

expand [–*tab***[,***tab***,***...* **]] [** *file ...* **]**

unexpand [–aA] [–*tab***[,***tab***,***...* **]] [** *file ...* **]**

DESCRIPTION

`expand` replaces each tab character (ASCII 9, control-I) found in each *file* with the correct number of spaces and writes the result to standard output. By default, tab positions are every four columns.

If a single argument of the form *-tab* is given, where *tab* is a decimal number, tab columns are taken to be every *tab* spaces. If a comma-separated list of values is given, the list specifies each actual tab column, which need not be evenly spaced. `expand` is useful for examining files created with soft tabs in a text editor if the logical tab size is different from the normal tab size of output devices (usually eight characters).

`unexpand` replaces with the correct number of tabs strings of two or more spaces, ending on tab columns, at the beginnings of lines. The *tab* size is given as with `expand`.

OPTIONS

–a `unexpand` will expand all strings of two or more spaces into tabs, not just the ones at the start of each line.

–A `unexpand` will expand all strings of one or more spaces into tabs.

SEE ALSO

Utility Programs - `pr`

NAME

expect – program a dialogue with interactive programs

SYNOPSIS

expect [-dDinN] [-c *cmds* **] [-[f|b]]** *cmdfile* **] [** *args* **]**

INTRODUCTION

Expect is a program that "talks" to other interactive programs according to a script. Following the script, Expect knows what can be expected from a program and what the correct response should be. An interpreted language provides branching and high-level control structures to direct the dialogue. In addition, the user can take control and interact directly when desired, afterward returning control to the script.

Expectk is a mixture of Expect and Tk. It behaves just like Expect and Tk's wish. Expect can also be used directly in C or C++ (that is, without Tcl). See libexpect(3).

The name "Expect" comes from the idea of *send/expect* sequences popularized by uucp, kermit and other modem control programs. However unlike uucp, Expect is generalized so that it can be run as a user-level command with any program and task in mind. Expect can actually talk to several programs at the same time.

For example, here are some things Expect can do:

- Cause your computer to dial you back, so that you can login without paying for the call.
- Start a game (e.g., rogue) and if the optimal configuration doesn't appear, restart it (again and again) until it does, then hand over control to you.
- Run fsck, and in response to its questions, answer "yes", "no" or give control back to you, based on predetermined criteria.
- Connect to another network or BBS (e.g., MCI Mail, CompuServe) and automatically retrieve your mail so that it appears as if it was originally sent to your local system.

- Carry environment variables, current directory, or any kind of information across `rlogin`, `telnet`, `tip`, `su`, `chgrp`, etc.

There are a variety of reasons why the shell cannot perform these tasks. (Try, you'll see.) All are possible with `Expect`.

In general, `Expect` is useful for running any program which requires interaction between the program and the user. All that is necessary is that the interaction can be characterized programmatically. `Expect` can also give the user back control (without halting the program being controlled) if desired. Similarly, the user can return control to the script at any time.

USAGE

`Expect` reads *cmdfile* for a list of commands to execute. `Expect` may also be invoked implicitly on systems which support the `#!` notation by marking the script executable, and making the first line in your script:

```
#!/usr/local/bin/expect -f
```

Of course, the path must accurately describe where `Expect` lives. `/usr/local/bin` is just an example.

The **-c** flag prefaces a command to be executed before any in the script. The command should be quoted to prevent being broken up by the shell. This option may be used multiple times. Multiple commands may be executed with a single **-c** by separating them with semicolons. Commands are executed in the order they appear. (When using Expectk, this option is specified as **-command**.)

The **-d** flag enables some diagnostic output, which primarily reports internal activity of commands such as `expect` and `interact`. This flag has the same effect as "`exp_internal 1`" at the beginning of an Expect script, plus the version of `Expect` is printed. (The `strace` command is useful for tracing statements, and the `trace` command is useful for tracing variable assignments.) (When using Expectk, this option is specified as **-diag**.)

The **-D** flag enables an interactive debugger. An integer value should follow. The debugger will take control before the next Tcl procedure if the value is non-zero or if a **^C** is pressed (or a breakpoint is hit, or other appropriate debugger command appears in the script). See the `README` file or SEE ALSO (below) for more information on the debugger. (When using Expectk, this option is specified as **-Debug**.)

The **-f** flag prefaces a file from which to read commands from. The flag itself is optional as it is only useful when using the #! notation (see above), so that other arguments may be supplied on the command line. (When using Expectk, this option is specified as **-file**.)

By default, the command file is read into memory and executed in its entirety. It is occasionally desirable to read files one line at a time. For example, stdin is read this way. In order to force arbitrary files to be handled this way, use the **-b** flag. (When using Expectk, this option is specified as **-buffer**.)

If the string "-" is supplied as a filename, standard input is read instead. (Use "`./-`" to read from a file actually named "-".)

The **-i** flag causes `Expect` to interactively prompt for commands instead of reading them from a file. Prompting is terminated via the `exit` command or upon EOF. See `interpreter` (below) for more information. **-i** is assumed if neither a command file nor **-c** is used. (When using Expectk, this option is specified as **-interactive**.)

-- may be used to delimit the end of the options. This is useful if you want to pass an option-like argument to your script without it being interpreted by `Expect`. This can usefully be placed in the `#!` line to prevent any flag-like interpretation by Expect. For example, the following will leave the original arguments (including the script name) in the variable `argv`.

```
#!/usr/local/bin/expect --
```

Note that the usual `getopt()` and `execve()` conventions must be observed when adding arguments to a `#!` line.

The file `$exp_library/expect.rc` is sourced automatically if present, unless the **-N** flag is used. (When using Expectk, this option is specified as **-NORC**.) Immediately after this, the file `~/.expect.rc` is sourced automatically, unless the **-n** flag is used. If the environment variable `DOTDIR` is defined, it is treated as a directory and `.expect.rc` is read from there. (When using Expectk, this option is specified as **-norc**.) This sourcing occurs only after executing any **-c** flags.

Optional *args* are constructed into a list and stored in the variable named `argv. argc` is initialized to the length of `argv`.

`argv0` is defined to be the name of the script (or binary if no script is used). For example, the following prints out the name of the script and the first three arguments:

```
send_user "$argv0 [lrange $argv 0 2]\n"
```

COMMANDS

`Expect` uses Tcl (Tool Command Language). Tcl provides control flow (e.g., if, for, break), expression evaluation and several other features such as recursion, procedure definition, etc. Commands used here but not defined (e.g., `set`, `if`, `exec`) are Tcl commands (see `tcl`(3)). `Expect` supports additional commands, described below. Unless otherwise specified, commands return the empty string.

Commands are listed alphabetically so that they can be quickly located. However, new users may find it easier to start by reading the descriptions of `spawn`, `send`, `expect`, and `interact`, in that order.

Note that the best introduction to the language (both Expect and Tcl) is provided in the book “Exploring Expect” (see SEE ALSO below). Examples are included in this man page but they are very limited since this man page is meant primarily as reference material.

Note that in the text of this man page, “Expect” with an uppercase “E” refers to the `Expect` program while “expect” with a lower-case “e” refers to the `expect` command within the `Expect` program.)

`close` [`-slave`] [`-onexec` 0|1] [`-i` *spawn_id*]

closes the connection to the current process. Most interactive programs will detect EOF on their stdin and exit; thus `close` usually suffices to kill the process as well. The **`-i`** flag declares the process to close corresponding to the named spawn id.

Both `expect` and `interact` will detect when the current process exits and implicitly do a `close`. But if you kill the process by, say, “exec kill $pid”, you will need to explicitly call `close`.

The **-onexec** flag determines whether the spawn id will be closed in any new spawned processes or if the process is overlayed. To leave a spawn id open, use the value 0. A non-zero integer value will force the spawn closed (the default) in any new processes.

The **-slave** flag closes the slave associated with the spawn id. (See "spawn **-pty**".) When the connection is closed, the slave is automatically closed as well if still open.

No matter whether the connection is closed implicitly or explicitly, you should call `wait` to clear up the corresponding kernel process slot. `close` does not call `wait` since there is no guarantee that closing a process connection will cause it to exit. See `wait` below for more info.

`debug` [[`-now`] `0|1`]

controls a Tcl debugger allowing you to step through statements, set breakpoints, etc.

With no arguments, a 1 is returned if the debugger is not running, otherwise a 0 is returned.

With a 1 argument, the debugger is started. With a 0 argument, the debugger is stopped. If a 1 argument is preceded by the **-now** flag, the debugger is started immediately (i.e., in the middle of the `debug` command itself). Otherwise, the debugger is started with the next Tcl statement.

The `debug` command does not change any traps. Compare this to starting Expect with the **-D** flag (see above).

See the README file or SEE ALSO (below) for more information on the debugger.

`disconnect` disconnects a forked process from the terminal. It continues running in the background. The process is given its own process group (if possible). Standard I/O is redirected to `/dev/null`.

The following fragment uses disconnect to continue running the script in the background.

```
if [fork]!=0 exit
 disconnect
 . . .
```

The following script reads a password, and then runs a program every hour that demands a password each time it is run. The script supplies the password so that you only have to type it once. (See the stty command which demonstrates how to turn off password echoing.)

```
send_user "password?\ "
 expect_user -re "(.*)\n"
 for {} 1 {} {
       if [fork]!=0 {sleep 3600;continue}
       disconnect
       spawn priv_prog
       expect Password:
       send "$expect_out(1,string)\r"
       . . .
       exit
 }
```

An advantage to using disconnect over the shell asynchronous process feature (&) is that Expect can save the terminal parameters prior to disconnection, and then later apply them to new ptys. With &, Expect does not have a chance to read the terminal's parameters since the terminal is already disconnected by the time Expect receives control.

exit [-*opts*] [*status*]

causes Expect to exit or otherwise prepare to do so.

The **-onexit** flag causes the next argument to be used as an exit handler. Without an argument, the current exit handler is returned.

The **-noexit** flag causes Expect to prepare to exit but stop short of actually returning control to the operating system. The user-defined exit handler is run as well as Expect's own internal handlers. No further Expect commands should be executed. This is useful if you are running Expect with other Tcl extensions. The current interpreter (and main window if in the Tk environment) remain so that other Tcl extensions can clean up. If Expect's `exit` is called again (however this might occur), the handlers are not rerun.

Upon exiting, all connections to spawned processes are closed. Closure will be detected as an EOF by spawned processes. `exit` takes no other actions beyond what the normal `_exit`(2) procedure does. Thus, spawned processes that do not check for EOF may continue to run. (A variety of conditions are important to determining, for example, what signals a spawned process will be sent, but these are system-dependent, typically documented under `exit`(3).) Spawned processes that continue to run will be inherited by `init`.

status (or 0 if not specified) is returned as the exit status of `Expect`. `exit` is implicitly executed if the end of the script is reached.

`exp_continue` The command `exp_continue` allows `expect` itself to continue executing rather than returning as it normally would. (See `expect` for more information.)

`exp_internal` [`-f` *file*] *value*

causes further commands to send diagnostic information internal to `Expect` to stderr if *value* is non-zero. This output is disabled if *value* is 0. The diagnostic information includes every character received, and every attempt made to match the current output against the patterns.

If the optional *file* is supplied, all normal and debugging output is written to that file (regardless of the value of *value*). Any previous diagnostic output file is closed.

The **-info** flag causes `exp_internal` to return a description of the most recent non-info arguments given.

`exp_open` [*args*] [`-i` *spawn_id*]

returns a Tcl file identifier that corresponds to the original spawn id. The file identifier can then be used as if it were opened by Tcl's `open` command. (The spawn id should no longer be used. A `wait` should not be executed.

The **-leaveopen** flag leaves the spawn id open for access through Expect commands. A `wait` must be executed on the spawn id.

`exp_pid` [`-i` *spawn_id*]
returns the process id corresponding to the currently spawned process. If the **-i** flag is used, the pid returned corresponds to that of the given spawn id.

`exp_send` is an alias for `send`.

`exp_send_error`
is an alias for `send_error`.

`exp_send_log` is an alias for `send_log`.

`exp_send_tty` is an alias for `send_tty`.

`exp_send_user` is an alias for `send_user`.

`exp_version` [[`-exit`] *version*]
is useful for assuring that the script is compatible with the current version of Expect.

With no arguments, the current version of `Expect` is returned. This version may then be encoded in your script. If you actually know that you are not using features of recent versions, you can specify an earlier version.

Versions consist of three numbers separated by dots. First is the major number. Scripts written for versions of `Expect` with a different major number will almost certainly not work. `exp_version` returns an error if the major numbers do not match.

Second is the minor number. Scripts written for a version with a greater minor number than the current version may depend upon some new feature and might not run. `exp_version` returns an error if the major numbers match, but the script minor number is greater than that of the running `Expect`.

Third is a number that plays no part in the version comparison. However, it is incremented when the `Expect` software distribution is changed in any way, such as by additional documentation or optimization. It is reset to 0 upon each new minor version.

With the **-exit** flag, `Expect` prints an error and exits if the version is out of date.

`expect` [[-*opts*] *pat1 body1*] ... [-*opts*] *patn* [*bodyn*]
waits until one of the patterns matches the output of a spawned process, a specified time period has passed, or an end-of-file is seen. If the final body is empty, it may be omitted.

Patterns from the most recent `expect_before` command are implicitly used before any other patterns. Patterns from the most recent `expect_after` command are implicitly used after any other patterns.

If the arguments to the entire `expect` statement require more than one line, all the arguments may be "braced" into one so as to avoid terminating each line with a backslash. In this one case, the usual Tcl substitutions will occur despite the braces.

If a pattern is the keyword `eof`, the corresponding body is executed upon end-of-file. If a pattern is the keyword `timeout`, the corresponding body is executed upon timeout. If no timeout keyword is used, an implicit null action is executed upon timeout. The default timeout period is 10 seconds but may be set, for example to 30, by the command "set timeout 30". An infinite timeout may be designated by the value `-1`. If a pattern is the keyword `default`, the corresponding body is executed upon either timeout or end-of-file.

If a pattern matches, then the corresponding body is executed. `expect` returns the result of the body (or the empty string if no pattern matched). In the event that multiple patterns match, the one appearing first is used to select a body.

Each time new output arrives, it is compared to each pattern in the order they are listed. Thus, you may test for absence of a match by making the last pattern something guaranteed to appear, such as a prompt. In situations where there is no prompt, you must use `timeout` (just like you would if you were interacting manually).

Patterns are specified in three ways. By default, patterns are specified as with Tcl's `string match` command. (Such patterns are also similar to C-shell regular expressions usually referred to as "glob" patterns). The **`-gl`** flag may may be used to protect patterns that might otherwise match `expect` flags from doing so. Any pattern beginning with a "-" should be protected this way. (All strings starting with "-" are reserved for future options.)

For example, the following fragment looks for a successful login. (Note that `abort` is presumed to be a procedure defined elsewhere in the script.)

```
expect {  busy {print busy\n ; exp_continue}
failed abort
"invalid password" abort
timeout abort
connected
     }
```

Quotes are necessary on the fourth pattern since it contains a space, which would otherwise separate the pattern from the action. Patterns with the same action (such as the 3rd and 4th) require listing the actions again. This can be avoid by using regexp-style patterns (see below). More information on forming glob-style patterns can be found in the Tcl manual.

Regexp-style patterns follow the syntax defined by Tcl's `regexp` (short for "regular expression") command. regexp patterns are introduced with the flag **-re**. The previous example can be rewritten using a regexp as:

```
expect {  busy {print busy\n ; exp_continue}
-re "failed|invalid password" abort
timeout abort
connected
      }
```

Both types of patterns are "unanchored". This means that patterns do not have to match the entire string, but can begin and end the match anywhere in the string (as long as everything else matches). Use ^ to match the beginning of a string, and $ to match the end. Note that if you do not wait for the end of a string, your responses can easily end up in the middle of the string as they are echoed from the spawned process. While still producing correct results, the output can look unnatural. Thus, use of $ is encouraged if you can exactly describe the characters at the end of a string.

Note that in many editors, the ^ and `$` match the beginning and end of lines respectively. However, because expect is not line oriented, these characters match the beginning and end of the data (as opposed to lines) currently in the expect matching buffer. (Also, see the note below on "system indigestion.")

The **-ex** flag causes the pattern to be matched as an *exact* string. No interpretation of `*`, `^`, etc is made (although the usual Tcl conventions must still be observed). Exact patterns are always unanchored.

The **-nocase** flag causes uppercase characters of the output to compare as if they were lowercase characters. The pattern is not affected.

While reading output, more than 2000 bytes can force earlier bytes to be "forgotten". This may be changed with the function `match_max`. (Note that excessively large values can slow down the pattern matcher.) If *patlist* is `full_buffer`, the corresponding body is executed if *match_max* bytes have been received and no other patterns have matched. Whether or not the `full_buffer` keyword is used, the forgotten characters are written to `expect_out(buffer)`.

If *patlist* is the keyword `null`, and nulls are allowed (via the `remove_nulls` command), the corresponding body is executed if a single ASCII 0 is matched. It is not possible to match 0 bytes via glob or regexp patterns.

Upon matching a pattern (or `eof` or `full_buffer`), any matching and previously unmatched output is saved in the variable `expect`. Up to 9 regexp substring matches are saved in the variables `expect_out(1,`*string*`)` through `expect_out(9,`*string*`)`. If the **`-indices`** flag is used before a pattern, the starting and ending indices (in a form suitable for `lrange`) of the 10 strings are stored in the variables `expect_out(`*X*`,start)` and `expect_out(`*X*`,end)` where X is a digit, corresponds to the substring position in the buffer. 0 refers to strings which matched the entire pattern and is generated for glob patterns as well as regexp patterns. For example, if a process has produced output of "`abcdefgh\n`", the result of:

```
expect "cd"
```

is as if the following statements had executed:

```
set expect_out(0,string) cd
set expect_out(buffer) abcd
```

and "`efgh\n`" is left in the output buffer. If a process produced the output "`abbbcabkkkka\n`", the result of:

```
expect -indices -re "b(b*).*(k+)"
```

is as if the following statements had executed:

```
set expect_out(0,start) 1
set expect_out(0,end) 10
set expect_out(0,string) bbbcabkkkk
set expect_out(1,start) 2
set expect_out(1,end) 3
set expect_out(1,string) bb
set expect_out(2,start) 10
```

```
set expect_out(2,end) 10
set expect_out(2,string) k
set expect_out(buffer) abbbcabkkkk
```

and "`a\n`" is left in the output buffer. The pattern "`*`" (and `-re` "`.*`") will flush the output buffer without reading any more output from the process.

Normally, the matched output is discarded from Expect's internal buffers. This may be prevented by prefixing a pattern with the **`-notransfer`** flag. This flag is especially useful in experimenting (and can be abbreviated to "**`-n`**" for convenience while experimenting).

The spawn id associated with the matching output (or `eof` or `full_buffer`) is stored in `expect`.

The **`-timeout`** flag causes the current expect command to use the following value as a timeout instead of using the value of the timeout variable.

By default, patterns are matched against output from the current process, however the **`-i`** flag declares the output from the named *spawn_id* list be matched against any following patterns (up to the next **`-i`**). The *spawn_id* list should either be a whitespace separated list of spawn ids or a variable referring to such a list of spawn ids.

For example, the following example waits for "connected" from the current process, or "busy", "failed" or "invalid password" from the spawn id named by `$proc2`.

```
expect {
  -i $proc2 busy{print busy\n ; exp_continue}
  -re "failed|invalid password" abort
  timeoutabort
  connected
  }
```

The value of the global variable `any_spawn_id` may be used to match patterns to any spawn_ids that are named with all other **`-i`** flags in the current `expect` command. The spawn id from a **`-i`** flag with no associated pattern (i.e., followed immediately by another **-i**) is made available to any other patterns in the same `expect` command associated with `any_spawn_id`.

The **-i** flag may also name a global variable in which case the variable is read for a list of spawn ids. The variable is reread whenever it changes. This provides a way of changing the I/O source while the command is in execution. Spawn ids provided this way are called "indirect" spawn ids.

Actions such as `break` and `continue` cause control structures (i.e., `for`, `proc`) to behave in the usual way. The command `exp_continue` allows `expect` itself to continue executing rather than returning as it normally would.

This is useful for avoiding explicit loops or repeated expect statements. The following example is part of a fragment to automate `rlogin`. The `exp_continue` avoids having to write a second `expect` statement (to look for the prompt again) if the `rlogin` prompts for a password.

```
expect {
Password: {
stty -echo
send_user "password (for $user) on $host: "
expect_user -re "(.*)\n"
send_user "\n"
send "$expect_out(1,string)\r"
stty echo
exp_continue
} incorrect {
send_user "invalid password or account\n"
exit
} timeout {
send_user "connection to $host timed out\n"
exit
} eof {
            send_user \
                "connection to host failed: $expect"
```

```
exit
} -re $prompt
}
```

For example, the following fragment might help a user guide an interaction that is already totally automated. In this case, the terminal is put into raw mode. If the user presses "+", a variable is incremented. If "p" is pressed, several returns are sent to the process, perhaps to poke it in some way, and "i" lets the user interact with the process, effectively stealing away control from the script. In each case, the `exp_continue` allows the current `expect` to continue pattern matching after executing the current action.

```
stty raw -echo
expect_after {
-i $user_spawn_id
"p" {send "\r\r\r"; exp_continue}
"+" {incr foo; exp_continue}
"i" {interact; exp_continue}
"quit" exit
}
```

By default, `exp_continue` resets the timeout timer. The timer is not restarted, if `exp_continue` is called with the **`-continue_timer`** flag.

`expect_after` [*expect_args*]

works identically to the `expect_before` except that if patterns from both `expect` and `expect_after` can match, the `expect` pattern is used. See the `expect_before` command for more information.

`expect_background` [*expect_args*]

takes the same arguments as `expect`, however it returns immediately. Patterns are tested whenever new input arrives. Note: this only works in the Tk environment. The pattern `timeout` and `default` are meaningless to `expect_background` and are silently discarded. Otherwise, the `expect_background` command uses `expect_before` and `expect_after` patterns just like `expect` does.

When `expect_background` actions are being evaluated, background processing for the same spawn id is blocked. Background processing is unblocked when the action completes. While background processing is blocked, it is possible to do a (foreground) `expect` on the same spawn id.

It is not possible to execute an `expect` while an `expect_background` is unblocked. `expect_background` for a particular spawn id is deleted by declaring a new expect_background with the same spawn id. Declaring `expect_background` with no pattern removes the given spawn id from the ability to match patterns in the background.

`expect_before` [*expect_args*]

takes the same arguments as `expect`, however it returns immediately. Pattern-action pairs from the most recent `expect_before` with the same spawn id are implicitly added to any following `expect` commands. If a pattern matches, it is treated as if it had been specified in the `expect` command itself, and the associated body is executed in the context of the `expect` command. If patterns from both `expect_before` and `expect` can match, the `expect_before` pattern is used.

If no pattern is specified, the spawn id is not checked for any patterns.

Unless overridden by a **-i** flag, `expect_before` patterns match against the spawn id defined at the time that the `expect_before` command was executed (not when its pattern is matched).

The **-info** flag causes `expect_before` to return the current specifications of what patterns it will match. By default, it reports on the current spawn id. An optional spawn id specification may be given for information on that spawn id. For example

```
expect_before -info -i $proc
```

At most one spawn id specification may be given. The flag **-indirect** suppresses direct spawn ids that come only from indirect specifications.

Instead of a spawn id specification, the flag "**-all**" will cause "-info" to report on all spawn ids.

The output of the **-info** flag can be reused as the argument to expect_before.

`expect_tty` [*expect_args*]
is like `expect` but it reads characters from `/dev/tty` (i.e. keystrokes from the user). By default, reading is performed in cooked mode. Thus, lines must end with a return in order for `expect` to see them. This may be changed via `stty` (see the `stty` command below).

`expect_user` [*expect_args*]
is like `expect` but it reads characters from stdin (i.e. keystrokes from the user). By default, reading is performed in cooked mode. Thus, lines must end with a return in order for `expect` to see them. This may be changed via `stty` (see the `stty` command below).

`fork` creates a new process. The new process is an exact copy of the current `Expect` process. On success, `fork` returns 0 to the new (child) process and returns the process ID of the child process to the parent process. On failure (invariably due to lack of resources, e.g., swap space, memory), `fork` returns -1 to the parent process, and no child process is created.

Forked processes exit via the `exit` command, just like the original process. Forked processes are allowed to write to the log files. If you do not disable debugging or logging in most of the processes, the result can be confusing.

Some pty implementations may be confused by multiple readers and writers, even momentarily. Thus, it is safest to `fork` before spawning processes.

`interact` [*string1 body1*] ... [*stringn* [*bodyn*]]
gives control of the current process to the user, so that keystrokes are sent to the current process, and the stdout and stderr of the current process are returned.

String-body pairs may be specified as arguments, in which case the body is executed when the corresponding string is entered. (By default, the string is not sent to the current process.) The `interpreter` command is assumed, if the final body is missing.

If the arguments to the entire `interact` statement require more than one line, all the arguments may be "braced" into one so as to avoid terminating each line with a backslash. In this one case, the usual Tcl substitutions will occur despite the braces.

For example, the following command runs interact with the following string-body pairs defined: When ^Z is pressed, `Expect` is suspended. (The **`-reset`** flag restores the terminal modes.) When ^A is pressed, the user sees "`you typed a control-A`" and the process is sent a ^A. When $ is pressed, the user sees the date. When ^C is pressed, `Expect` exits. If "`foo`" is entered, the user sees "`bar`". When ~~ is pressed, the `Expect` interpreter runs interactively.

```
set CTRLZ \032
interact {
-reset $CTRLZ {exec kill -STOP [pid]}
\001{send_user "you typed a control-A\n";
send "\001"
}
${send_user "The date is [exec date]."}
\003exit
foo{send_user "bar"}
~~
    }
```

In string-body pairs, strings are matched in the order they are listed as arguments. Strings that partially match are not sent to the current process in anticipation of the remainder coming. If characters are then entered such that there can no longer possibly be a match, only the part of the string will be sent to the process that cannot possibly begin another match. Thus, strings that are substrings of partial matches can match later, if the original strings that was attempting to be match ultimately fails.

By default, string matching is exact with no wild cards. (In contrast, the `expect` command uses glob-style patterns by default.) The **-ex** flag may be used to protect patterns that might otherwise match `interact` flags from doing so. Any pattern beginning with a "-" should be protected this way. (All strings starting with "-" are reserved for future options.)

The **-re** flag forces the string to be interpreted as a regexp-style pattern. In this case, matching substrings are stored in the variable `interact_out` similarly to the way `expect` stores its output in the variable `expect_out`. The **-indices** flag is similarly supported.

The pattern `eof` introduces an action that is executed upon end-of-file. A separate `eof` pattern may also follow the **-output** flag in which case it is matched if an `eof` is detected while writing output. The default `eof` action is "return", so that `interact` simply returns upon any EOF.

The pattern `timeout` introduces a timeout (in seconds) and action that is executed after no characters have been read for a given time. The `timeout` pattern applies to the most recently specified process. There is no default timeout. The special variable "`timeout`" (used by the `expect` command) has no affect on this timeout.

For example, the following statement could be used to autologout users who have not typed anything for an hour but who still get frequent system messages:

```
interact -input $user_spawn_id timeout 3600 \
return -output $spawn_id
```

If the pattern is the keyword `null`, and nulls are allowed (via the `remove_nulls` command), the corresponding body is executed if a single ASCII 0 is matched. It is not possible to match 0 bytes via glob or regexp patterns.

Prefacing a pattern with the flag **-iwrite** causes the variable *expect* to be set to the spawn_id which matched the pattern (or `eof`).

Actions such as `break` and `continue` cause control structures (i.e., `for`, `proc`) to behave in the usual way. However `return` causes interact to return to its caller, while `inter_return` causes `interact` to cause a return in its caller. For example, if "proc foo" called `interact` which then executed the action `inter_return`, `proc foo` would return. (This means that if `interact` calls `interpreter` interactively typing `return` will cause the interact to continue, while `inter_return` will cause the interact to return to its caller.)

During `interact`, raw mode is used so that all characters may be passed to the current process. If the current process does not catch job control signals, it will stop if sent a stop signal (by default ^Z). To restart it, send a continue signal (such as by "`kill -CONT` *<pid>*"). If you really want to send a SIGSTOP to such a process (by ^Z), consider spawning csh first and then running your program. On the other hand, if you want to send a SIGSTOP to `Expect` itself, first press the escape character, and then press ^Z.

String-body pairs can be used as a shorthand for avoiding having to enter the interpreter and execute commands interactively. The previous terminal mode is used while the body of a string-body pair is being executed.

For speed, actions execute in raw mode by default. The **`-reset`** flag resets the terminal to the mode it had before `interact` was executed (invariably, cooked mode). Note that characters entered when the mode is being switched may be lost (an unfortunate feature of the terminal driver on some systems). The only reason to use **`-reset`** is if your action depends on running in cooked mode.

The **`-echo`** flag sends characters that match the following pattern back to the process that generated them as each character is read. This may be useful when the user needs to see feedback from partially typed patterns.

If a pattern is being echoed but eventually fails to match, the characters are sent to the spawned process. If the spawned process then echoes them, the user will see the characters twice. **-echo** is probably only appropriate in situations where the user is unlikely to not complete the pattern. For example, the following excerpt is from `rftp`, the recursive-ftp script, where the user is prompted to enter ~g, ~p, or ~l, to get, put, or list the current directory recursively. These are so far away from the normal ftp commands, that the user is unlikely to type ~ followed by anything else, except mistakenly, in which case, they'll probably just ignore the result anyway.

```
interact {
-echo ~g {getcurdirectory 1}
-echo ~l {getcurdirectory 0}
-echo ~p {putcurdirectory}
    }
```

The **-nobuffer** flag sends characters that match the following pattern on to the output process as characters are read.

This is useful when you wish to let a program echo back the pattern. For example, the following might be used to monitor where a person is dialing (a Hayes-style modem). Each time "`atd`" is seen the script logs the rest of the line.

```
proc lognumber {} {
  interact -nobuffer -re "(.*)\r" return
  puts $log "[exec date]: dialed
$interact_out(1,string)"
}
interact -nobuffer "atd" lognumber
```

During `interact`, previous use of `log_user` is ignored. In particular, `interact` will force its output to be logged (sent to the standard output) since it is presumed the user doesn't wish to interact blindly.

The **-o** flag causes any following key-body pairs to be applied to the output of the current process. This can be useful, for example, when dealing with hosts that send unwanted characters during a telnet session.

By default, `interact` expects the user to be writing stdin and reading stdout of the `Expect` process itself. The **-u** flag (for "user") makes `interact` look for the user as the process named by its argument (which must be a spawned id).

This allows two unrelated processes to be joined together without using an explicit loop. To aid in debugging, Expect diagnostics always go to stderr (or stdout for certain logging and debugging information). For the same reason, the `interpreter` command will read interactively from stdin.

For example, the following fragment creates a login process. Then it dials the user (not shown), and finally connects the two together. Of course, any process may be substituted for login. A shell, for example, would allow the user to work without supplying an account and password.

```
spawn login
 set login $spawn_id
 spawn tip modem
 # dial back out to user
 # connect user to login
 interact -u $login
```

To send output to multiple processes, list each spawn id list prefaced by a **-output** flag. Input for a group of output spawn ids may be determined by a spawn id list prefaced by a **-input** flag. (Both **-input** and **-output** may take lists in the same form as the **-i** flag in the `expect` command, except that `any_spawn_id` is not meaningful in `interact`.) All following flags and strings (or patterns) apply to this input until another **-input** flag appears. If no **-input** appears, **-output** implies "`-input $user_spawn_id -output`". (Similarly, with patterns that do not have **-input**.) If one **-input** is specified, it overrides `$user_spawn_id`. If a second **-input** is specified, it overrides `$spawn_id`. Additional **-input** flags may be specified.

The two implied input processes default to having their outputs specified as `$spawn_id` and `$user_spawn_id` (in reverse). If a **-input** flag appears with no **-output** flag, characters from that process are discarded.

The **-i** flag introduces a replacement for the current `spawn_id` when no other **-input** or **-output** flags are used.

It is possible to change the processes that are being interacted with by using indirect spawn ids. (Indirect spawn ids are described in the section on the expect command.) Indirect spawn ids may be specified with the **-i**, **-u**, **-input**, or **-output** flags.

`interpreter` causes the user to be interactively prompted for `Expect` and Tcl commands. The result of each command is printed.

Actions such as `break` and `continue` cause control structures (i.e., `for`, `proc`) to behave in the usual way. However `return` causes interpreter to return to its caller, while `inter_return` causes `interpreter` to cause a return in its caller. For example, if "`proc foo`" called `interpreter` which then executed the action `inter_return`, `proc foo` would return. Any other command causes `interpreter` to continue prompting for new commands.

By default, the prompt contains two integers. The first integer describes the depth of the evaluation stack (i.e., how many times `Tcl_Eval` has been called). The second integer is the Tcl history identifier. The prompt can be set by defining a procedure called "`prompt1`" whose return value becomes the next prompt. If a statement has open quotes, parens, braces, or brackets, a secondary prompt (by default "+>") is issued upon newline. The secondary prompt may be set by defining a procedure called "`prompt2`".

During `interpreter`, cooked mode is used, even if the its caller was using raw mode.

`log_file` [*args*] [[-a] *file*]

If a file name is provided, `log_file` will record a transcript of the session (beginning at that point) in the file. `log_file` will stop recording if no argument is given. Any previous log file is closed.

Instead of a file name, a Tcl file identifier may be provided by using the **-open** or **-leaveopen** flags. This is similar to the `spawn` command. (See `spawn` for more info.)

The **-a** flag forces output to be logged that was suppressed by the `log_user` command.

By default, the `log_file` command *appends* to old files rather than truncating them, for the convenience of being able to turn logging off and on multiple times in one session. To truncate files, use the **-noappend** flag.

The **-info** flag causes log_file to return a description of the most recent non-info arguments given.

`log_user` -info|0|1

By default, the send/expect dialogue is logged to stdout (and a logfile if open). The logging to stdout is disabled by the command "`log_user 0`" and reenabled by "`log_user 1`". Logging to the logfile is unchanged.

The **-info** flag causes log_user to return a description of the most recent non-info arguments given.

`match_max` [-d] [-i *spawn_id*] [*size*]

defines the size of the buffer (in bytes) used internally by `expect`. With no *size* argument, the current size is returned.

With the **-d** flag, the default size is set. (The initial default is 2000.) With the **-i** flag, the size is set for the named spawn id, otherwise it is set for the current process.

`overlay` [-# *spawn_id*] [-# *spawn_id*] [...] *program* [*args*]

executes *program args* in place of the current `Expect` program, which terminates. A bare hyphen argument forces a hyphen in front of the command name as if it was a login shell. All spawn_ids are closed except for those named as arguments. These are mapped onto the named file identifiers.

Spawn_ids are mapped to file identifiers for the new program to inherit. For example, the following line runs `chess` and allows it to be controlled by the current process - say, a chess master.

```
overlay -0 $spawn_id -1 $spawn_id -2 $spawn_id
chess
```

This is more efficient than "`interact -u`", however, it sacrifices the ability to do programmed interaction since the `Expect` process is no longer in control.

Note that no controlling terminal is provided. Thus, if you disconnect or remap standard input, programs that do job control (shells, `login`, etc) will not function properly.

`parity` [-d] [-i *spawn_id*] [*value*]
defines whether parity should be retained or stripped from the output of spawned processes. If *value* is zero, parity is stripped, otherwise it is not stripped. With no *value* argument, the current value is returned.

With the **-d** flag, the default parity value is set. (The initial default is 1, i.e., parity is not stripped.) With the **-i** flag, the parity value is set for the named spawn id, otherwise it is set for the current process.

`remove_nulls` [-d] [-i *spawn_id*] [*value*]
defines whether nulls are retained or removed from the output of spawned processes before pattern matching or storing in the variable `expect_out` or `interact_out`. If *value* is 1, nulls are removed. If *value* is 0, nulls are not removed. With no *value* argument, the current value is returned.

With the **-d** flag, the default value is set. (The initial default is 1, i.e., nulls are removed.) With the **-i** flag, the value is set for the named spawn id, otherwise it is set for the current process.

Whether or not nulls are removed, `Expect` will record null bytes to the log and stdout.

`send [-`*flags*`]` *string*

Sends *string* to the current process. For example, the command

```
send "hello world\r"
```

sends the characters, h e l l o <blank> w o r l d <return> to the current process. (Tcl includes a printf-like command (called `format`) which can build arbitrarily complex strings.)

Characters are sent immediately although programs with line-buffered input will not read the characters until a return character is sent. A return character is denoted "\r".

The **--** flag forces the next argument to be interpreted as a string rather than a flag. Any string can be preceded by "**--**" whether or not it actually looks like a flag. This provides a reliable mechanism to specify variable strings without being tripped up by those that accidentally look like flags. (All strings starting with "-" are reserved for future options.)

The **-i** flag declares that the string be sent to the named *spawn_id*. If the *spawn_id* is `user_spawn_id`, and the terminal is in raw mode, newlines in the string are translated to return-newline sequences so that they appear as it the terminal was in cooked mode. The **-raw** flag disables this translation.

The **-null** flag sends null characters (0 bytes). By default, one null is sent. An integer may follow the **-null** to indicate how many nulls to send.

The **-break** flag generates a break condition. This only makes sense if the spawn id refers to a tty device opened via "`spawn -open`". If you have spawned a process such as tip, you should use tip's convention for generating a break.

The **-s** flag forces output to be sent *slowly*, thus avoid the common situation where a computer outtypes an input buffer that was designed for a human who would never outtype the same buffer. This output is controlled by the value of the variable "`send_slow`" which takes a two element list. The first element is an integer that describes the number of bytes to send atomically. The second element is a real number that describes the number of seconds by which the atomic sends must be separated. For example, "`set send_slow {10 .001}`" would force "`send -s`" to send strings with 1 millisecond in between each 10 characters sent.

The **-h** flag forces output to be sent (somewhat) like a human actually typing. Human-like delays appear between the characters. (The algorithm is based upon a Weibull distribution, with modifications to suit this particular application.) This output is controlled by the value of the variable "`send_human`" which takes a five element list. The first two elements are average interarrival time of characters in seconds. The first is used by default. The second is used at word endings, to simulate the subtle pauses that occasionally occur at such transitions. The third parameter is a measure of variability where .1 is quite variable, 1 is reasonably variable, and 10 is quite invariable. The extremes are 0 to infinity. The last two parameters are, respectively, a minimum and maximum interarrival time. The minimum and maximum are used last and "clip" the final time. The ultimate average can be quite different from the given average if the minimum and maximum clip enough values.

As an example, the following command emulates a fast and consistent typist:

```
set send_human {.1 .3 1 .05 2}
send -h "I'm hungry.  Let's do lunch."
```

while the following might be more suitable after a hangover:

```
set send_human {.4 .4 .2 .5 100}
send -h "Goodd party lash night!"
```

Note that errors are not simulated, although you can set up error correction situations yourself by embedding mistakes and corrections in a send argument.

The flags for sending null characters, for sending breaks, for forcing slow output and for human-style output are mutually exclusive. Only the one specified last will be used. Furthermore, no *string* argument can be specified with the flags for sending null characters or breaks.

It is a good idea to precede the first `send` to a process by an `expect`. `expect` will wait for the process to start, while `send` cannot. In particular, if the first `send` completes before the process starts running, you run the risk of having your data ignored. In situations where interactive programs offer no initial prompt, you can precede `send` by a delay as in:

```
# To avoid giving hackers hints on how to break in,
# this system does not prompt for an external
password.
# Wait for 5 seconds for exec to complete
spawn telnet very.secure.gov
sleep 5
send password\r
```

`exp_send` is an alias for `send`. If you are using Expectk or some other variant of Expect in the Tk environment, `send` is defined by Tk for an entirely different purpose. `exp_send` is provided for compatibility between environments. Similar aliases are provided for other Expect's other send commands.

`send_error` [`-flags`] string
: is like `send`, except that the output is sent to stderr rather than the current process.

`send_log` [\--] string
: is like `send`, except that the string is only sent to the log file (see `log_file`.) The arguments are ignored if no log file is open.

`send_tty` [`-flags`] string
is like `send`, except that the output is sent to `/dev/tty` rather than the current process.

`send_user` [`-flags`] string
is like `send`, except that the output is sent to stdout rather than the current process.

`sleep` *seconds* causes the script to sleep for the given number of seconds. Seconds may be a decimal number. Interrupts (and Tk events if you are using Expectk) are processed while Expect sleeps.

`spawn` [*args*] *program* [*args*]
creates a new process running *program args*. Its stdin, stdout and stderr are connected to Expect, so that they may be read and written by other `Expect` commands. The connection is broken by `close` or if the process itself closes any of the file identifiers.

When a process is started by `spawn`, the variable `spawn_id` is set to a descriptor referring to that process. The process described by `spawn_id` is considered the current process. `spawn_id` may be read or written, in effect providing job control.

`user_spawn_id` is a global variable containing a descriptor which refers to the user. For example, when `spawn_id` is set to this value, `expect` behaves like `expect_user`.

`error_spawn_id` is a global variable containing a descriptor which refers to the standard error. For example, when `spawn_id` is set to this value, `send` behaves like `send_error`.

`tty_spawn_id` is a global variable containing a descriptor which refers to `/dev/tty`. If `/dev/tty` does not exist (such as in a cron, at, or batch script), then `tty_spawn_id` is not defined. This may be tested as:

```
if [info vars tty_spawn_id] {
# /dev/tty exists
} else {
```

```
# /dev/tty doesn't exist
# probably in cron, batch, or at script
}
```

`spawn` returns the UNIX process id. If no process is spawned, 0 is returned. The variable `spawn_out(slave,name)` is set to the name of the pty slave device.

By default, `spawn` echoes the command name and arguments. The **-noecho** flag stops `spawn` from doing this.

The **-console** flag causes console output to be redirected to the spawned process. This is not supported on all systems.

Internally, `spawn` uses a pty, initialized the same way as the user's tty. This is further initialized so that all settings are "`sane`" (according to `stty`(1)). If the variable `stty_init` is defined, it is interpreted in the style of `stty` arguments as further configuration. For example, "`set stty_init raw`" will cause further spawned processes's terminals to start in raw mode. **-nottycopy** skips the initialization based on the user's tty. **-nottyinit** skips the "`sane`" initialization.

Normally, `spawn` takes little time to execute. If you notice `spawn` taking a significant amount of time, it is probably encountering ptys that are wedged. A number of tests are run on ptys to avoid entanglements with errant processes. (These take 10 seconds per wedged pty.) Running Expect with the **-d** option will show if `Expect` is encountering many ptys in odd states. If you cannot kill the processes to which these ptys are attached, your only recourse may be to reboot.

If *program* cannot be spawned successfully because `exec`(2) fails (e.g. when *program* doesn't exist), an error message will be returned by the next `interact` or `expect` command as if *program* had run and produced the error message as output. This behavior is a natural consequence of the implementation of `spawn`. Internally, `spawn` forks, after which the spawned process has no way to communicate with the original `Expect` process except by communication via the spawn id.

The **-open** flag causes the next argument to be interpreted as a Tcl file identifier (i.e., returned by `open`.) The spawn id can then be used as if it were a spawned process. (The file identifier should no longer be used.) This lets you treat raw devices, files, and pipelines as spawned processes without using a pty. 0 is returned to indicate there is no associated process. When the connection to the spawned process is closed, so is the Tcl file identifier. The **-leaveopen** flag is similar to **-open** except that **-leaveopen** causes the file identifier to be left open even after the spawn id is closed.

The **-pty** flag causes a pty to be opened but no process spawned. 0 is returned to indicate there is no associated process. Spawn_id is set as usual.

The variable `spawn_out(slave,fd)` is set to a file identifier corresponding to the pty slave. It can be closed using "close **-slave**".

The **-ignore** flag names a signal to be ignored in the spawned process. Otherwise, signals get the default behavior. Signals are named as in the `trap` command, except that each signal requires a separate flag.

`strace` *level* causes following statements to be printed before being executed. (Tcl's trace command traces variables.) *level* indicates how far down in the call stack to trace. For example, the following command runs `Expect` while tracing the first 4 levels of calls, but none below that.

```
expect -c "strace 4" script.exp
```

The **-info** flag causes strace to return a description of the most recent non-info arguments given.

`stty` *args* changes terminal modes similarly to the external stty command.

By default, the controlling terminal is accessed. Other terminals can be accessed by appending "`< /dev/tty...`" to the command. (Note that the arguments should not be grouped into a single argument.)

Requests for status return it as the result of the command. If no status is requested and the controlling terminal is accessed, the previous status of the raw and echo attributes are returned in a form which can later be used by the command.

For example, the arguments `raw` or **`-cooked`** put the terminal into raw mode. The arguments **`-raw`** or `cooked` put the terminal into cooked mode. The arguments `echo` and **`-echo`** put the terminal into echo and noecho mode respectively.

The following example illustrates how to temporarily disable echoing. This could be used in otherwise-automatic scripts to avoid embedding passwords in them. (See more discussion on this under EXPECT HINTS below.)

```
stty -echo
send_user "Password: "
expect_user -re "(.*)\n"
set password $expect_out(1,string)
stty echo
```

`system` *args* — gives *args* to `sh`(1) as input, just as if it had been typed as a command from a terminal. `Expect` waits until the shell terminates. The return status from sh is handled the same way that `exec` handles its return status.

In contrast to `exec` which redirects stdin and stdout to the script, `system` performs no redirection (other than that indicated by the string itself). Thus, it is possible to use programs which must talk directly to `/dev/tty`. For the same reason, the results of `system` are not recorded in the log.

`timestamp` [*args*]

returns a timestamp. With no arguments, the number of seconds since the epoch is returned.

The **-format** flag introduces a string which is returned but with substitutions made according to the POSIX rules for `strftime()`. For example %a is replaced by an abbreviated weekday name (i.e., Sat). Others are:

```
%aabbreviated weekday name
%Afull weekday name
%babbreviated month name
%Bfull month name
%cdate-time as in: Wed Oct  6 11:45:56 1993
%dday of the month (01-31)
%Hhour (00-23)
%Ihour (01-12)
%jday (001-366)
%mmonth (01-12)
%Mminute (00-59)
%pam or pm
%Ssecond (00-61)
%uday (1-7, Monday is first day of week)
%Uweek (00-53, first Sunday is first day of week
one)
%Vweek (01-53, ISO 8601 style)
%wday (0-6)
%Wweek (00-53, first Monday is first day of week
one)
%xdate-time as in: Wed Oct  6 1993
%Xtime as in: 23:59:59
%yyear (00-99)
%Yyear as in: 1993
%Ztimezone (or nothing if not determinable)
%%a bare percent sign
```

Other % specifications are undefined. Other characters will be passed through untouched. Only the C locale is supported.

The **-seconds** flag introduces a number of seconds since the epoch to be used as a source from which to format. Otherwise, the current time is used.

The **-gmt** flag forces `timestamp` output to use the GMT timezone. With no flag, the local timezone is used.

`trap` [[*command*] *signals*]

causes the given *command* to be executed upon future receipt of any of the given signals. The command is executed in the global scope. If *command* is absent, the signal action is returned. If *command* is the string `SIG_IGN`, the signals are ignored. If *command* is the string `SIG_DFL`, the signals are result to the system default. *signals* is either a single signal or a list of signals. Signals may be specified numerically or symbolically as per `signal`(3). The "`SIG`" prefix may be omitted.

With no arguments (or the argument **-number),** `trap` returns the signal number of the trap command currently being executed.

The **-code** flag uses the return code of the command in place of whatever code Tcl was about to return when the command originally started running.

The **-interp** flag causes the command to be evaluated using the interpreter active at the time the command started running rather than when the trap was declared.

The **-name** flag causes the `trap` command to return the signal name of the trap command currently being executed.

The **-max** flag causes the `trap` command to return the largest signal number that can be set.

For example, the command "`trap {send_user "Ouch!"} SIGINT`" will print "`Ouch!`" each time the user presses ^C.

By default, `SIGINT` (which can usually be generated by pressing ^C) and `SIGTERM` cause Expect to exit. This is due to the following trap, created by default when Expect starts.

```
trap exit {SIGINT SIGTERM}
```

If you use the **-D** flag to start the debugger, `SIGINT` is redefined to start the interactive debugger. This is due to the following trap:

```
trap {exp_debug 1} SIGINT
```

The debugger trap can be changed by setting the environment variable `EXPECT_DEBUG_INIT` to a new trap command.

You can, of course, override both of these just by adding trap commands to your script. In particular, if you have your own "`trap exit SIGINT`", this will override the debugger trap. This is useful if you want to prevent users from getting to the debugger at all.

If you want to define your own trap on SIGINT but still trap to the debugger when it is running, use:

```
if ![exp_debug] {trap mystuff SIGINT}
```

Alternatively, you can trap to the debugger using some other signal.

`trap` will not let you override the action for `SIGALRM` as this is used internally to `Expect`. The disconnect command sets `SIGALRM` to `SIG_IGN` (ignore). You can reenable this as long as you disable it during subsequent `spawn` commands.

See `signal`(3) for more info.

`wait` [*args*] delays until a spawned process (or the current process if none is named) terminates.

`wait` normally returns a list of four integers. The first integer is the pid of the process that was waited upon. The second integer is the corresponding spawn id. The third integer is `-1` if an operating system error occurred, or `0` otherwise. If the third integer was `0`, the fourth integer is the status returned by the spawned process. If the third integer was `-1`, the fourth integer is the value of errno set by the operating system. The global variable `errorCode` is also set.

Additional elements may appear at the end of the return value from `wait`. An optional fifth element identifies a class of information. Currently, the only possible value for this element is `CHILDKILLED` in which case the next two values are the C-style signal name and a short textual description.

The **-i** flag declares the process to wait corresponding to the named *spawn_id* (NOT the process id). It is possible to wait for any spawned process by using the `spawn id -1`.

The **-nowait** flag causes the wait to return immediately with the indication of a successful wait. When the process exits (later), it will automatically disappear without the need for an explicit wait.

LIBRARIES

Expect automatically knows about two built-in libraries for Expect scripts. These are defined by the directories named in the variables `exp_library` and `exp_exec_library`. Both are meant to contain utility files that can be used by other scripts.

`exp_library` contains architecture-independent files. `exp_exec_library` contains architecture-dependent files. Depending on your system, both directories may be totally empty. The existence of the file `$exp_exec_library/cat-buffers` describes whether your `/bin/cat` buffers by default.

PRETTY-PRINTING

A `vgrind` definition is available for pretty-printing `Expect` scripts. Assuming the `vgrind` definition supplied with the `Expect` distribution is correctly installed, you can use it as:

```
vgrind -lexpect file
```

EXAMPLES

It many not be apparent how to put everything together that the man page describes. I encourage you to read and try out the examples in the example directory of the `Expect` distribution. Some of them are real programs. Others are simply illustrative of certain techniques, and of course, a couple are just quick hacks. The `INSTALL` file has a quick overview of these programs.

The `Expect` papers (see SEE ALSO) are also useful. While some papers use syntax corresponding to earlier versions of Expect, the accompanying rationales are still valid and go into a lot more detail than this man page.

CAVEATS

Extensions may collide with Expect's command names. For example, `send` is defined by Tk for an entirely different purpose. For this reason, most of the `Expect` commands are also available as "`exp_XXXX`". Commands and variables beginning with "`exp`", "`inter`", "`spawn`", and "`timeout`" do not have aliases. Use the extended command names if you need this compatibility between environments.

`Expect` takes a rather liberal view of scoping. In particular, variables read by commands specific to the `Expect` program will be sought first from the local scope, and if not found, in the global scope. For example, this obviates the need to place "`global timeout`" in every procedure you write that uses `expect`. On the other hand, variables written are always in the local scope (unless a "`global`" command has been issued). The most common problem this causes is when `spawn` is executed in a procedure. Outside the procedure, *spawn_id* no longer exists, so the spawned process is no longer accessible simply because of scoping. Add a "`global spawn_id`" to such a procedure.

If you cannot enable the multispawning capability (i.e., your system supports neither select (BSD *.*), poll (SVR>2), nor something equivalent), `Expect` will only be able to control a single process at a time. In this case, do not attempt to set `spawn_id`, nor should you execute processes via exec while a spawned process is running. Furthermore, you will not be able to `expect` from multiple processes (including the user as one) at the same time.

Terminal parameters can have a big effect on scripts. For example, if a script is written to look for echoing, it will misbehave if echoing is turned off. For this reason, Expect forces sane terminal parameters by default. Unfortunately, this can make things unpleasant for other programs. As an example, the emacs shell wants to change the "usual" mappings: newlines get mapped to newlines instead of carriage-return newlines, and echoing is disabled. This allows one to use emacs to edit the input line. Unfortunately, Expect cannot possibly guess this.

You can request that Expect not override its default setting of terminal parameters, but you must then be very careful when writing scripts for such environments. In the case of emacs, avoid depending upon things like echoing and end-of-line mappings.

The commands that accepted arguments braced into a single list (the `expect` variants and `interact`) use a heuristic to decide if the list is actually one argument or many. The heuristic can fail only in the case when the list actually does represent a single argument which has multiple embedded newlines with non-whitespace characters between them. This seems sufficiently improbable, however the argument "-brace" can be used to force a single argument to be handled as a single argument. This could conceivably be used with machine-generated Expect code.

BUGS

It was really tempting to name the program "sex" (for either "Smart EXec" or "Send-EXpect"), but good sense (or perhaps just Puritanism) prevailed.

On some systems, when a shell is spawned, it complains about not being able to access the tty but runs anyway. This means your system has a mechanism for gaining the controlling tty that `Expect` doesn't know about. Please find out what it is, and send this information back to me.

Ultrix 4.1 (at least the latest versions around here) considers timeouts of above 1000000 to be equivalent to 0.

IRIX 6.0 does not handle pty permissions correctly so that if Expect attempts to allocate a pty previously used by someone else, it fails. Upgrade to IRIX 6.1.

Telnet (verified only under SunOS 4.1.2) hangs if `TERM` is not set. This is a problem under cron and at, which do not define `TERM`. Thus, you must set it explicitly even though it is usually irrelevant. It just has to be set to something! The following probably suffices for most cases.

```
setenv(TERM) vt100
```

Some implementations of ptys are designed so that the kernel throws away any unread output after 10 to 15 seconds (actual number is implementation-dependent) after the process has closed the file descriptor. Thus `Expect` programs such as

```
spawn date
sleep 20
expect
```

will fail. To avoid this, invoke non-interactive programs with `exec` rather than `spawn`. While such situations are conceivable, in practice I have never encountered a situation in which the final output of a truly interactive program would be lost due to this behavior.

On the other hand, Cray UNICOS ptys throw away any unread output immediately after the process has closed the file descriptor. I have reported this to Cray and they are working on a fix.

Sometimes a delay is required between a prompt and a response, such as when a tty interface is changing UART settings or matching baud rates by looking for start/stop bits. Usually, all this is require is to sleep for a second or two. A more robust technique is to retry until the hardware is ready to receive input. The following example uses both strategies:

```
send "speed 9600\r";
sleep 1
expect {
timeout {send "\r"; exp_continue}
$prompt
}
```

EXPECT HINTS

There are a couple of things about `Expect` that may be non-intuitive. This section attempts to address some of these things with a couple of suggestions.

A common expect problem is how to recognize shell prompts. Since these are customized differently by differently people and different shells, portably automating `rlogin` can be difficult without knowing the prompt. A reasonable convention is to have users store a regular expression describing their prompt (in particular, the end of it) in the environment variable EXPECT_PROMPT. Code like the following can be used. If EXPECT_PROMPT doesn't exist, the code still has a good chance of functioning correctly.

```
set prompt "(%|#|\\$) $"          ;# default prompt
catch {set prompt $env(EXPECT_PROMPT)}

expect -re $prompt
```

I encourage you to write expect patterns that include the end of whatever you expect to see. This avoids the possibility of answering a question before seeing the entire thing. In addition, while you may well be able to answer questions before seeing them entirely, if you answer early, your answer may appear echoed back in the middle of the question. In other words, the resulting dialogue will be correct but look scrambled.

Most prompts include a space character at the end. For example, the prompt from ftp is 'f', 't', 'p', '>' and <blank>. To match this prompt, you must account for each of these characters. It is a common mistake not to include the blank. Put the blank in explicitly.

If you use a pattern of the form X*, the * will match all the output received from the end of X to the last thing received. This sounds intuitive but can be somewhat confusing because the phrase "last thing received" can vary depending upon the speed of the computer and the processing of I/O both by the kernel and the device driver.

In particular, humans tend to see program output arriving in huge chunks (atomically) when in reality most programs produce output one line at a time. Assuming this is the case, the * in the pattern of the previous paragraph may only match the end of the current line even though there seems to be more, because at the time of the match that was all the output that had been received.

expect has no way of knowing that further output is coming unless your pattern specifically accounts for it.

Even depending on line-oriented buffering is unwise. Not only do programs rarely make promises about the type of buffering they do, but system indigestion can break output lines up so that lines break at seemingly random places. Thus, if you can express the last few characters of a prompt when writing patterns, it is wise to do so.

If you are waiting for a pattern in the last output of a program and the program emits something else instead, you will not be able to detect that with the `timeout` keyword. The reason is that `expect` will not timeout; instead it will get an `eof` indication. Use that instead. Even better, use both. That way if that line is ever moved around, you won't have to edit the line itself.

Newlines are usually converted to carriage return, linefeed sequences when output by the terminal driver. Thus, if you want a pattern that explicitly matches the two lines, from, say, `printf("foo\nbar")`, you should use the pattern "`foo\r\nbar`".

A similar translation occurs when reading from the user, via `expect_user`. In this case, when you press return, it will be translated to a newline. If `Expect` then passes that to a program which sets its terminal to raw mode (like telnet), there is going to be a problem, as the program expects a true return. (Some programs are actually forgiving in that they will automatically translate newlines to returns, but most don't.) Unfortunately, there is no way to find out that a program put its terminal into raw mode.

Rather than manually replacing newlines with returns, the solution is to use the command "`stty raw`", which will stop the translation. Note, however, that this means that you will no longer get the cooked line-editing features.

`interact` implicitly sets your terminal to raw mode so this problem will not arise then.

It is often useful to store passwords (or other private information) in `Expect` scripts. This is not recommended since anything that is stored on a computer is susceptible to being accessed by anyone. Thus, interactively prompting for passwords from a script is a smarter idea than embedding them literally. Nonetheless, sometimes such embedding is the only possibility.

Unfortunately, the UNIX file system has no direct way of creating scripts which are executable but unreadable. Systems which support setgid shell scripts may indirectly simulate this as follows:

Create the `Expect` script (that contains the secret data) as usual. Make its permissions be 750 (`-rwxr-x---`) and owned by a trusted group, i.e., a group which is allowed to read it. If necessary, create a new group for this purpose. Next, create a `/bin/sh` script with permissions 2751 (`-rwxr-s--x`) owned by the same group as before.

The result is a script which may be executed (and read) by anyone. When invoked, it runs the `Expect` script.

SEE ALSO

Library Functions - `Tcl`, `libexpect`

Documents - *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs* by Don Libes.

Articles - "expect: Curing Those Uncontrollable Fits of Interactivity" by Don Libes, *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June 11-15, 1990.

"Using `expect` to Automate System Administration Tasks" by Don Libes, *Proceedings of the 1990 USENIX Large Installation Systems Administration Conference*, Colorado Springs, Colorado, October 17-19, 1990.

"Tcl: An Embeddable Command Language" by John Ousterhout, *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 22-26, 1990.

"expect: Scripts for Controlling Interactive Programs" by Don Libes, *Computing Systems*, Vol. 4,

No. 2, University of California Press Journals, November 1991.

"Regression Testing and Conformance Testing Interactive Programs", by Don Libes, *Proceedings of the Summer 1992 USENIX Conference*, pp. 135-144, San Antonio, TX, June 12-15, 1992.

"Kibitz - Connecting Multiple Interactive Programs Together", by Don Libes, *Software - Practice & Experience*, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May, 1993.

"A Debugger for Tcl Applications", by Don Libes, *Proceedings of the 1993 Tcl/Tk Workshop*, Berkeley, CA, June 10-11, 1993.

AUTHOR

Don Libes, National Institute of Standards and Technology

ACKNOWLEDGEMENTS

Thanks to John Ousterhout for Tcl, and Scott Paisley for inspiration. Thanks to Rob Savoye for Expect's autoconfiguration code.

The HISTORY file documents much of the evolution of `expect`. It makes interesting reading and might give you further insight to this software. Thanks to the people mentioned in it who sent me bug fixes and gave other assistance.

Design and implementation of `Expect` was paid for in part by the U.S. government and is therefore in the public domain. However the author and NIST would like credit if this program and documentation or portions of them are used.

NAME

`expectk` – Expect with Tk support

SYNOPSIS

expectk [*args* **]**

INTRODUCTION

`Expectk` is a combination of Expect with Tk. (See their respective man pages for a more comprehensive explanation of either.) `Expectk` should run any `wish` or `Expect` script (with minor changes – see below).

The differences between the Expectk and Expect environment follows.

The `send` command is Tk's. Expect's `send` command can be invoked by the name `exp_send`. (For compatibility, Expect allows either `send` or `exp_send` to be used.)

Scripts may be invoked implicitly on systems which support the `#!` notation by marking the script executable, and making the first line in your script:

```
#!/usr/local/bin/expectk -f
```

Of course, the path must accurately describe where `Expectk` lives. `/usr/local/bin` is just an example.

NAME

`exportfs` – export and unexport directories to NFS clients

SYNOPSIS

exportfs [-avui] [-o *options* **] [** *directory* **]**

DESCRIPTION

`exportfs` makes a local directory available for mounting over the network by NFS clients. It is normally invoked at boot time by the `/net/rc.network` script, and uses information contained in the `/etc/exports` file to export a **`directory`** (which must be specified as a full pathname). The super-user can run `exportfs` at any time to alter the list or characteristics of exported directories. Directories that are currently exported are listed in the file `/etc/xtab`. With no options or arguments, `exportfs` prints out the list of directories currently exported.

OPTIONS

-a All. Export all directories listed in `/etc/exports`, or if **-u** is specified, unexport all of the currently exported directories.

-v Verbose. Print each directory as it is exported or unexported.

-u Unexport the indicated directories.

-i Ignore the options in `/etc/exports`. Normally, `exportfs` will consult `/etc/exports` for the options associated with the exported directory.

-o *options* Specify a comma-separated list of optional characteristics for the directory being exported. *options* can be selected from among:

ro Export the directory read-only. If not specified, the directory is exported read-write.

rw=*hostname[:hostname]..*
Export the directory read-mostly. Read-mostly means exported read-only to most

machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

anon=*uid* If a request comes from an unknown user, use `uid` as the effective user ID. Note: root users (uid 0) are always considered “unknown” by the NFS server, unless they are included in the “root” options below. The default value for this option is **−2**.

root=*hostname[:hostname]..*
Give root access only to the root users from a specified *hostname*. The default is for no hosts to be granted root access.

access=*hostname[:hostname]...*
Give mount access to each *hostname* listed. Each *hostname* in the list is checked in the `/etc/hosts` database. The default value allows any machine to mount the given directory.

NOTE

Unlike other versions of NFS you cannot mount subdirectories of exported directories, but the subdirectories can be exported and then mounted.

FILES

`/etc/exports`	static export information
`/etc/xtab`	current state of exported directories

SEE ALSO

File Formats - `exports`

NAME

`expr` – expression evaluator and pattern matcher

SYNOPSIS

`expr` *expression*

DESCRIPTION

`expr` interprets its arguments as an expression, which it then evaluates. The result is written to standard output.

SYNTAX

Each individual command line argument is taken as a distinct token. Special characters must be protected from interpretation from command line interpreters.

An argument not matching a known operator is taken to be a string. Certain operators interpret strings as numeric values. The operators recognized by `expr` are listed below in order of decreasing precedence.

`e1 : e2` Expression `e1` is compared as a string to `e2`. The second expression is interpreted as a regular expression, with the following meta-characters recognized:

- *a* Matches single character *a*.
- *.* Matches any single character (except null).
- *[list]* Matches any character in *list*. A range of characters is denoted by *a-b*.
- *[^list]* Matches any character not in the *list*.
- *a** Matches zero or more instances of *a*.
- *ab* Matches a followed by *b*.

Any meta-character may be quoted with a backslash (which itself may need to be quoted to hide it from a shell). A portion of the pattern expression may be grouped with \(...\) to cause a portion of the matched string `e1` to be used as the result of the operator. If no portion of e2 is so grouped, the : operator results in the number of characters matched (0 if the match failed).

`e1 * e2`
`e1 / e2`
`e1 % e2`
	Expressions `e1` and `e2` are interpreted as (possibly signed) integers. The result is the product, quotient, or remainder of the expressions.

`e1 + e2` `e1 - e2`	Expressions `e1` and `e2` are interpreted as (possibly signed) integers. The result is the sum or difference between the values.

`e1 > e2`
`e1 < e2`
`e1 >= e2`
`e1 <= e2`
`e1 = e2`
`e1 != e2`
	The expressions `e1` and `e2` are compared accordingly. The result is "1" if the comparison succeeds, or "0" if it fails. If both strings are numeric, comparison is numeric; otherwise comparison is lexicographic.

`e1 & e2`	If both `e1` and `e2` are non-null and non-zero (a string of any number of "0" characters), the result is `e1`; otherwise the result is "0".
`e1 | e2`	If `e1` is non-null and non-zero, the result is `e1`. If not, then if `e2` is non-null and non-zero, the result is `e2`. If neither expression is "true", the result is "0".

NOTE

Many characters used as operators must be quoted to prevent evaluation by the shell.

DIAGNOSTICS

`expr` exits with 0 if the result of the expression is neither null nor zero, or with 1 if the result is null or zero. If the expression contains a syntactic error, `expr` exits with 2.

SEE ALSO

Utility Programs - `test`

NAME

`ezsetup` – configure SMC Elite Series Ethernet cards (x86)

SYNOPSIS

`ezsetup`

DESCRIPTION

LynxOS x86 systems only.

`ezsetup` allows the user to configure the SMC Elite series (AT bus) Ethernet cards for LynxOS. This interactive utility will help the user to set up the following software-configurable parameters.

```
I/O Base Address
IRQ
RAM Size
RAM Base Address
Add Wait States
Network Connection
ROM Size
ROM Base Address
```

All of these parameters can be configured only with the "soft jump" option. Please refer to the corresponding adapter manual for more information on jumper settings.

The values used by LynxOS for the above parameters are:

I/O Base Address	0x240
IRQ	5
RAM Size	(Unaltered; use the default value itself)
RAM Base Address	0xCC000
Add Wait States	No
Network Connection	(Use the appropriate option AUI/BNC/10BaseT)

ROM Size	Disabled
ROM Base Address	Disabled

Once these values are chosen and saved from within `ezsetup`, the card becomes usable under LynxOS. On subsequent reboot/reset conditions, these values remain configured in the card.

In case of diskless LynxOS x86 clients, the card should be configured to enable the ROM segment by choosing ROM Size as 64KB and ROM Base Address as 0xD0000, in addition to the above standard values. Once the LynxOS Netboot PROM is installed in the PROM slot, the control will come to the Netboot PROM on every reset/PowerOn.

ACKNOWLEDGEMENTS

This software was developed by the Standard Microsystems Corporation.

NAME

`factor` – resolve an integer into its prime factors

SYNOPSIS

factor [-l] [*number*]

DESCRIPTION

The `factor` utility resolves an integer into its prime factors. Without an argument, `factor` reads lines from its standard input. If it reads a non-zero integer with magnitude less than or equal to 2^{53}, it will resolve it into its prime factors and print each factor the proper number of times. If it reads a non-integral number or a number that is too large, `factor` will print an appropriate message. If it reads zero or encounters any character that is inconsistent with a numeric quantity, `factor` will exit.

If invoked with an argument, `factor` will resolve the integer into its prime factors, as described above, and then exit.

Depending on the number, `factor` can sometimes take a long time to complete the factorization, with the worst case occurring when the number is prime or the square of a prime. In such cases, the time taken is proportional to the square root of the number.

OPTIONS

-l Display the output on a single line rather than each divisor on its own line. Different divisors are separated by the multiplication symbol (`*`), and multiple occurrences of the same divisor are indicated using the exponentiation symbol (`^`).

SEE ALSO

Utility Programs - `calc`

NAME

`false`, `true` – provide truth values

SYNOPSIS

`false`

`true`

DESCRIPTION

`true` and `false` do nothing, and return 0 and 1, respectively. They are useful in writing Bourne shell scripts.

EXAMPLES

```
while true
do
  command list
done
```

SEE ALSO

Utility Programs - `sh`

NAME

fgrep – search a file for a string

SYNOPSIS

fgrep [–[[AB]]*num***] –***num* **[–[CVbchilnsvwxy]] [–[ef]]** *expr* **[***files...***]**

DESCRIPTION

fgrep searches the input *files* (the standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. fgrep patterns are fixed strings; no regular expression metacharacters are supported.

Take care when using the characters $, *, [, ^, |, (,), and \ in the *expr*, as these characters are also meaningful to the shell. It is safest to enclose the entire *expr* argument in single quotes ('...').

When any of the grep utilities is applied to more than one input file, the name of the file is displayed preceding each line which matches the pattern. The file name is not displayed when processing a single file, so if you actually want the file name to appear, use /dev/null as a second file in the list.

OPTIONS

–*num*	Print *num* lines of context before and after matches
–b	Print byte offsets
–c	Suppress all normal output; print a count of matches
–e *expr*	Take *expr* as the expression to be matched
–f *file*	Read patterns from *file*
–h	Don't print file names
–i, **–y**	Match upper and lower case
–l	Suppress all normal output; print line numbers

-n	Print line numbers
-s	Suppress all normal output
-v	Print lines that don't match
-w	Match words
-x	Match lines
-A *num*	Print *num* lines of context after matches
-B *num*	Print *num* lines of context before matches
-C	Print 2 lines of context before and after matches
-V	Print `fgrep` version number

COPYRIGHT

SEE ALSO

Utility Programs - `gawk, sed, sh, vi`

NAME

`file` – classify files

SYNOPSIS

`file [-cfm]` *file ...*

DESCRIPTION

`file` examines each file named and attempts to determine its nature. The file names and types are printed to standard output.

If the file is not a regular file, the type is determined from the mode. If the file is a regular file with an identifiable magic number, the type is given by that number (object files, executable files, and scripts for interpreters such as the shell). If the file contains mostly printable characters, `file` performs some pattern matching, and the type will be TEX or LATEX source, C source, English text, or just ASCII data.

OPTIONS

`-c` Decode the `/etc/magic` file and print to `stdout`.

`-f` *namefile* Check the files named in *namefile*.

`-m` *magicfile* Use *magicfile* instead of `/etc/magic` as the magic number file.

NOTE

The algorithm used by `file` is accurate enough to determine whether a file is printable, but it should not be relied upon in critical situations; it can be easily fooled.

SEE ALSO

Utility Programs - `ls, shownode`

File Formats - `a.out`

NAME

`find` – locate files

SYNOPSIS

find *directory* ... [*search-expression*]

DESCRIPTION

`find` searches the subtree beneath each *directory* for files which match the given *search-expression*. The *search-expression* is written in terms of file characteristics described below. Each file encountered in each subtree is processed through the expression.

Terms of the expression are evaluated until the truth or falsehood of the expression is known. Terms with side effects such as **-print** should be arranged in the expression such that the side effect occurs after any pre-qualifications for the file have been determined by other terms. If an expression does not include any side effects (no instances of **-print**, **-exec**, **-ls**, or **-ok**), `find` automatically adds a **-print** to the end of the expression.

Nearly every datum associated with a file can be checked with one of the primitive operators listed below. In the descriptions below, the argument *n* means exact comparison to a decimal integer; +*n* means comparison for greater than *n*; and -*n* means comparison for less than *n*.

OPTIONS

-atime *n*	True if the current file has been accessed *n* days ago.
-atime +*n*	True if the current file has been accessed more than +*n* days.
-atime -*n*	True if the current file has been accessed less than -*n* days.
-exec *command*	True if *command* executes and returns a zero status. The *command* may be a list of words terminated by a semicolon in a word by itself and protected from shell interpretation. The pathname under evaluation can be passed to *command* as the word “{}”.

-group *gname*	True if the current file belongs to the group *gname*. *gname* can be a group name or a numeric group ID.
-inum *n*	True if the current file has inode number *n*.
-inum +*n*	True if the current file has inode number greater than *n*.
-inum -*n*	True if the current file has inode number less than *n*.
-links *n*	True if the inode of the current file has *n* links.
-links +*n*	True if the inode of the current file has greater than *n* links.
-links -*n*	True if the inode of the current file has less than *n* links.
-ls **-lso** *flags*	Prints the current file name and relevant information about the file in the style of the **ls** program. The **-ls** version gives the mode, number of links, owner, size in bytes, modification time, and name. The **-lso** version allows some *flags* understood by `ls`, including **g** for group name, **i** for inode number, **s** for size in blocks, **c** for inode change time instead of modification time, **u** for access time instead of modification time, and **F** to append a file type indicator to each file name.
-mtime *n*	True if the current file has been changed *n* days ago.
-mtime +*n*	True if the current file has been changed more than +*n* days.
-mtime -*n*	True if the current file has been changed less than -*n* days.
-name *filename*	True if the *filename* argument matches the current file name. The normal `dlsh` wildcard syntax may be used, and must be quoted appropriately.
-newer *file*	True if the current file has been modified more recently than the argument *file*.
-ngroup	True if the group ID of the file is *not* recorded in the group database.

-nuser	True if the user ID of the file is *not* recorded in the password database.
-ok *command*	Basically the same as **-exec**, except that *command* is written to the standard output and executed only upon a response of y from the user.
-once	Instructs `find` to exit successfully as soon as a file is found that satisfies the rest of the expression. For example, if it is known that only one file satisfies the conditions, the **-once** will prevent `find` from searching further when the file is found. This flag always succeeds and so can be placed anywhere in the expression.
-perm *onum*	True if *onum* (an octal number) matches the current file's file permission flags (see chmod for more information on file permission flags). If *onum* is prefixed with a minus sign ("-"), then more flag bits become significant (see the description of the system call `stat`), and the flags are compared in the following way: *(flags & onum == onum)*
-print	Always true; the current filename is printed.
-size *n*	True if the current file is *n* blocks long.
-size +*n*	True if the current file is greater than +*n* blocks long.
-size -n	True if the current file is less than -*n* blocks long.
-type *c*	True if the current file type is *c*, where *c* can be one of the following:

b	block special file
c	character special file
d	directory file
-	normal file (AT&T uses **-f**)
r	normal file
f	normal file
p	named pipe (FIFO)
l	symbolic link

+	contiguous file
s	socket
i	non-persistent `ipc` special file
I	persistent `ipc` special file

–user *uname* True if *uname* is the owner of the file, where *uname* can be either the user's login name or his numeric user ID.

–xdev Forces `find` to stay within the device which contains each starting directory. Thus, `find` can be started somewhere within the root file system and instructed not to descend into any mounted file systems. The **–xdev** operator returns true and can thus be placed anywhere within the expression.

The basic terms can be combined into expressions with the following operators (listed in order of decreasing precedence):

1. Parentheses, "**(**", "**)**", to control the order of evaluation.
2. Exclamation point, "**!**", a unary operator which negates subexpressions.
3. Juxtaposing of terms implies logical *and*. Note that evaluation of the expression terminates as soon as the truth or falsehood of the expression is determined, so evaluation of a list of terms stops as soon a "false" one is found.
4. Terms connected by **–o** are logically OR'ed together. As with logical *and*, evaluation stops as soon as a "true" expression is found.

EXAMPLES

To remove all files "`xxx`" or "`yyy`" in the directory named "`fred`" which have not been accessed for three days:

```
find fred (–name xxx –o –name yyy) –atime +3 –exec rm {} \;
```

To `find` a file named "`xxx`" under the directory "`fred`":

```
find fred –name xxx –print
```

FILES

`/etc/passwd`	User name database
`/etc/group`	Group names

SEE ALSO

Utility Programs - `dlsh, ls, shownode`

System Calls – `stat()`

Library Functions - `directory`

NAME

`finger` – user information lookup program

SYNOPSIS

`finger [`*options***`]`** *name ...*

DESCRIPTION

By default `finger` lists the login name, full name, terminal name and write status (as a "`*`" before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current LynxOS user. (Idle time is minutes if it is a single integer; hours and minutes if a "`:`" is present; or days and hours if a "`d`" is present.)

A longer format also exists and is used by `finger` whenever a list of people's names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file `.plan` in their home directory, and the project on which they are working from the file `.project` also in the home directory.

`finger` may be used to lookup users on a remote machine. The format is to specify the user as `user@host`. If the user name is left off, the standard format listing is provided on the remote machine.

OPTIONS

`finger` options include:

`-b` Suppress printing the user's home directory and shell in a long format printout.

`-f` Suppress printing the header that is normally printed in a non-long format printout.

`-h` Suppress printing of the `.project` file in a long format printout.

-i	Force "idle" output format, which is similar to short format, except that only the login name, terminal, login time, and idle time are printed.
-l	Force long output format.
-m	Match arguments only on user name (not first or last name).
-p	Suppress printing of the `.plan` file in a long format printout.
-q	Force quick output format, which is similar to short format, except that only the login name, terminal, and login time are printed.
-s	Force short output format.
-w	Suppress printing the full name in a short format printout.

FILES

`/etc/inetd.conf`	Current inetd configuration file.
`/etc/utmp`	Current user file
`/etc/passwd`	For users names, offices, etc.
`/usr/adm/lastlog`	Last login times
`/.plan`	Plans
`/.project`	Projects

NOTE

Only the first line of the `.project` file is printed.

ACKNOWLEDGEMENTS

This section was developed by the University of California, Berkeley.

SEE ALSO

Utility Programs - `passwd, who, inetd, fingerd`

NAME

`fingerd` – remote user information server

SYNOPSIS

`fingerd`

DESCRIPTION

`fingerd` is a simple protocol based on RFC742 that provides an interface to the Name and Finger programs at several network sites. The program is supposed to return a friendly, human-oriented status report on either the system at the moment or a particular person in depth. There is no required format and the protocol consists mostly of specifying a single “command line”.

`fingerd` listens for TCP requests at port 79. Once connected, it reads a single command line terminated by a `<CRLF>` which is passed to `finger`.

`fingerd` closes its connections as soon as the output is finished.

If the line is null (i.e., just a `<CRLF>` is sent) then `finger` returns a “default” report that lists all people logged into the system at that moment.

If a user name is specified (e.g., eric `<CRLF>`), the response lists more extended information for only that particular user, whether logged in or not. Allowable “names” in the command line include both “login names” and “user names”. If a name is ambiguous, all possible derivations are returned.

ACKNOWLEDGEMENTS

This section was developed by the University of California, Berkeley.

SEE ALSO

Utility Programs - `finger, inetd`

NAME

`fixup` – make a SPARC executable/bootable by OBP

SYNOPSIS

`fixup` *filename*

DESCRIPTION

`fixup` is used to change the first two bytes of the magic number to 0x103 so the SPARC OBP will boot it.

NAME

flex – fast lexical analyzer generator

SYNOPSIS

flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -o*output* **-P***prefix* **-S***skeleton***] [--help --version] [***filename ...***]**

DESCRIPTION

flex is a tool for generating *scanners,* programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules.* flex generates as output a C source file, lex.yy.c, which defines a routine yylex(). This file is compiled and linked with the **-lfl** library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

SOME SIMPLE EXAMPLES

First some simple examples to get the flavor of how one uses flex. The following flex input specifies a scanner which whenever it encounters the string username will replace it with the user's login name:

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a flex scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of username expanded. In this input, there is just one rule. username is the *pattern* and the printf is the *action.* The %% marks the beginning of the rules.

Here's another simple example:

```
int num_lines = 0, num_chars = 0;
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
```

```
%%
main()
     {
     yylex();
     printf( "# of lines = %d, # of chars = %d\n",
     num_lines, num_chars );
    }
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, `num_lines` and `num_chars`, which are accessible both inside `yylex()` and in the `main()` routine declared after the second `%%`. There are two rules, one which matches a newline (`\n`) and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the '.' regular expression).

A somewhat more complicated example:

```
/* scanner for a toy Pascal-like language */
%{
/* need this for the call to atof() below */
#include <math.h>
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+    {
printf( "An integer:%s (%d)\n",yytext,atoi(yytext));
}
{DIGIT}+"."{DIGIT}*        {
 printf( "A float: %s (%g)\n", yytext,atof( yytext));
}
if|then|begin|end|procedure|function        {
printf( "A keyword: %s\n", yytext );
}
{ID} printf( "An identifier: %s\n", yytext );
"+"|"-"|"*"|"/" printf( "An operator: %s\n", yytext );
"{"[^}\n]*"}"     /* eat up one-line comments */
[ \t\n]+          /* eat up whitespace */
. printf( "Unrecognized character: %s\n", yytext );
%%
main( argc, argv )
int argc;
char **argv;
{
++argv, --argc;  /* skip over program name */
if ( argc > 0 )
yyin = fopen( argv[0], "r" );
else
yyin = stdin;
yylex();
}
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of tokens and reports on what it has seen.

The details of this example will be explained in the following sections.

FORMAT OF THE INPUT FILE

The `flex` input file consists of three sections, separated by a line with just `%%` in it:

definitions
`%%`
rules
`%%`
user_code

The *definitions* section contains declarations of simple name definitions to simplify the scanner specification, and declarations of *start conditions,* which are explained in a later section.

Name definitions have the form:

name definition

The *name* is a word beginning with a letter or an underscore (_) followed by zero or more letters, digits, underscores, or hyphens. The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using {*name*}, which will expand to (*definition*). For example,

```
DIGIT    [0-9]
ID      [a-z][a-z0-9]*
```

defines `DIGIT` to be a regular expression which matches a single digit, and `ID` to be a regular expression which matches a letter followed by zero-or-more letters or digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a period (`.`) followed by zero-or-more digits.

The *rules* section of the `flex` input contains a series of rules of the form:

pattern *action*

where the *pattern* must be unindented and the *action* must begin on the same line.

See below for a further description of patterns and actions.

Finally, the *user_code* section is simply copied to `lex.yy.c` verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped, too.

In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with the `%{}` characters removed). The characters enclosed in `%{` and `}` must be unindented and must be placed on lines by themselves.

In the rules section, any indented or `%{}`-enclosed text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or `%{}` text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for POSIX compliance; see below for other such features).

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with `/*`) is also copied verbatim to the output up to the next `*/`.

PATTERNS

The patterns in the input are written using an extended set of regular expressions. These are:

x	match the character *x*
`.`	Any character except newline
`[xyz]`	A character class; in this case, the pattern matches either `x`, `y`, or `z`
`[abj-oZ]`	A character class with a range in it; matches `a`, `b`, any letter from `j` through `o`, or a `Z`

`[^A-Z]`	A negated character class; i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
`[^A-Z\n]`	Any character EXCEPT an uppercase letter or a newline
`r*`	Zero or more `r`'s, where `r` is any regular expression
`r+`	One or more `r`'s
`r?`	Zero or one `r` (that is, an optional `r`)
`r{2,5}`	Anywhere from two to five `r`'s
`r{2,}`	Two or more `r`'s
`r{4}`	Exactly four `r`'s
`{name}`	The expansion of the name definition (see above)
`"[xyz]\"foo"`	The literal string: `[xyz]"foo`
`\`*x*	If *x* is `a`, `b`, `f`, `n`, `r`, `t`, or `v`, then the ANSI-C interpretation of `\`*x*. Otherwise, a literal *x* character (used to escape operators such as `*`)
`\123`	The character with octal value `123`
`\x2a`	The character with hexadecimal value `2a`
`(r)`	Match an `r`; parentheses are used to override precedence (see below)
`rs`	The regular expression `r` followed by the regular expression `s`; called concatenation
`r|s`	Either an `r` or an `s`
`r/s`	An `r` but only if it is followed by an `s`. The text matched by `s` is included when determining whether this rule is the longest-match rule, but is then returned to the input before the action is executed. So the action only sees the text matched by `r`. This type of pattern is called trailing context. (There are some combinations of `r/s` that flex cannot match correctly; see notes in the Deficiencies/Bugs section below regarding `dangerous trailing context` errors.)

`^r`	An `r`, but only at the beginning of a line
`r$`	An `r`, but only at the end of a line. Equivalent to `r/\n`.
`<s>r`	An `r`, but only in start condition `s` (see below for discussion of start conditions)
`<s1,s2,s3>r`	Same, but in any of start conditions `s1`, `s2`, or `s3`

`<<EOF>>` An end-of-file `<s1,s2><<EOF>>` an end-of-file when in start condition `s1` or `s2`

Note that inside of a character class, all regular expression operators lose their special meaning except escape ('\') and the character class operators, '–', ']', and, at the beginning of the class, '^'.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('|'). This pattern therefore matches *either* the string `foo` *or* the string `ba` followed by zero-or-more `r` characters. To match `foo` or zero-or-more `bar` sequences, use:

```
foo|(bar)*
```

and to match zero-or-more `foo` or `bar` sequences:

```
(foo|bar)*
```

In addition to characters and ranges of characters, character classes can also contain character class *expressions.* These are expressions enclosed inside `[:` and `:]` delimiters (which themselves must appear between the '`[`' and '`]`' of the character class; other elements may occur inside the character class, too). The valid expressions are:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

These expressions all designate a set of characters equivalent to the corresponding standard C `isXXX` function. For example, `[:alnum:]` designates those characters for which `isalnum()` returns true – i.e., any alphabetic or numeric. Some systems don't provide `isblank()`, so `flex` defines `[:blank:]` as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If your scanner is case-insensitive (the **-i** flag), then `[:upper:]` and `[:lower:]` are equivalent to `[:alpha:]`.

Some notes on patterns:

- A negated character class such as the example `[^A-Z]` above will match a newline unless `\n` (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., `[^A-Z\n]`). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like `[^"]*` can match the entire input unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '`/`' operator or the '`$`' operator). The start condition, '`^`', and `<<EOF>>` patterns can only occur at the beginning of a pattern, and, as well as with '`/`' and '`$`', cannot be grouped inside

parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

The following are illegal:

```
foo/bar$
<sc1>foo<sc2>bar
```

Note that the first of these, can be written `foo/bar\n`.

The following will result in '$' or '^' being treated as a normal character:

```
foo|(bar$)
foo|^bar
```

If what's wanted is a `foo` or a `bar`, followed by a newline, the following could be used (the special '|' action is explained below):

```
foo      |
bar$     /* action goes here */
```

A similar trick will work for matching a `foo` or a bar at the beginning of a line.

HOW THE INPUT IS MATCHED

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the `flex` input file is chosen.

Once the match is determined, the text corresponding to the match (called the *token)* is made available in the global character pointer `yytext,` and its length in the global integer `yyleng`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the *default rule* is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal `flex` input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

Note that `yytext` can be defined in two different ways: either as a character pointer or as a character array. You can control which definition `flex` uses by including one of the special directives `%pointer` or `%array` in the first (definitions) section of your `flex` input. The default is `%pointer`, unless you use the `-l lex` compatibility option, in which case `yytext` will be an array. The advantage of using `%pointer` is substantially faster scanning and no buffer overflow when matching very large tokens (unless you run out of dynamic memory). The disadvantage is that you are restricted in how your actions can modify `yytext` (see the next section), and calls to the `unput()` function destroys the present contents of `yytext`, which can be a considerable porting headache when moving between different `lex` versions.

The advantage of `%array` is that you can then modify `yytext` to your heart's content, and calls to `unput()` do not destroy `yytext` (see below). Furthermore, existing `lex` programs sometimes access `yytext` externally using declarations of the form:

```
extern char yytext[];
```

This definition is erroneous when used with `%pointer`, but correct for `%array`.

`%array` defines `yytext` to be an array of `YYLMAX` characters, which defaults to a fairly large value. You can change the size by simply using the `#define` directive to set `YYLMAX` to a different value in the first section of your `flex` input. As mentioned above, with `%pointer` yytext grows dynamically to accommodate large tokens. While this means your `%pointer` scanner can accommodate very large tokens (such as matching entire blocks of comments), bear in mind that each time the scanner must resize `yytext` it also must rescan the entire token from the beginning, so matching such tokens can prove slow. `yytext` presently does *not* dynamically grow if a call to `unput()` results in too much text being pushed back; instead, a run-time error results.

Also note that you cannot use `%array` with C++ scanner classes (the `c++` option; see below).

ACTIONS

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded. For example, here is the specification for a program which deletes all occurrences of `zap me` from its input:

```
%%
"zap me"
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$      /* ignore this token */
```

If the action contains a '`{`', then the action spans till the balancing '`}`' is found, and the action may cross multiple lines. `flex` knows about C strings and comments and won't be fooled by braces found within them, but also allows actions to begin with `%{` and will consider the action to be all the text up to the next `%}` (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('`|`') means *same as the action for the next rule*. See below for an illustration.

Actions can include arbitrary C code, including `return` statements to return a value to whatever routine called `yylex()`. Each time `yylex()` is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a `return`.

Actions are free to modify `yytext` except for lengthening it (adding characters to its end because doing so will overwrite later characters in the input stream). This however does not apply when using `%array` (see above); in that case, `yytext` may be freely modified in any way.

Actions are free to modify `yyleng` except they should not do so if the action also includes use of `yymore()` (see below).

There are a number of special directives which can be included within an action:

`ECHO` copies `yytext` to the scanner's output.

`BEGIN` followed by the name of a start condition places the scanner in the corresponding start condition (see below).

`REJECT` directs the scanner to proceed on to the *second best* rule which matched the input (or a prefix of the input). The rule is chosen as described above in the section entitled How the Input is Matched, and `yytext` and `yyleng` set up appropriately. It may either be one which matched as much text as the originally chosen rule but came later in the `flex` input file, or one which matched less text. For example, the following will both count the words in the input and call the routine `special()` whenever `frob` is seen:

```
int word_count = 0;
%%
frob        special(); REJECT;
[^ \t\n]+   ++word_count;
```

Without the `REJECT`, any occurrences of `frob` in the input would not be counted as words, since the scanner normally executes only one action per token. Multiple `REJECT` directives are allowed, each one finding the next best choice to the currently active rule. For example, when the following scanner scans the token `abcd`, it will write `abcdabcaba` to the output:

```
%%
a         |
ab        |
abc       |
abcd      ECHO; REJECT;
.|\n      /* eat up any unmatched character */
```

(The first three rules share the fourth's action since they use the special '|' action.) `REJECT` is a particularly expensive feature in terms of scanner performance; if it is used in *any* of the scanner's actions it will slow down *all* of the scanner's matching. Furthermore, `REJECT` cannot be used with the **`-Cf`** or **`-CF`** options (see below).

Note also that unlike the other special actions, `REJECT` is a branch; code immediately following it in the action will *not* be executed.

`yymore()` tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of `yytext` rather than replacing it. For example, given the input `mega-kludge` the following will write `mega-mega-kludge` to the output:

```
%%
mega-     ECHO; yymore();
kludge    ECHO;
```

First `mega-` is matched and echoed to the output. Then `kludge` is matched, but the previous `mega-` is still hanging around at the beginning of `yytext` so the `ECHO` for the `kludge` rule will actually write `mega-kludge`.

Two notes regarding use of `yymore()`. First, `yymore()` depends on the value of `yyleng` correctly reflecting the size of the current token, so you must not modify yyleng if you are using `yymore()`. Second, the presence of `yymore()` in the scanner's action entails a minor performance penalty in the scanner's matching speed.

`yyless(n)` returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. `yytext` and `yyleng` are adjusted appropriately (e.g., `yyleng` will now be equal to *n*). For example, on the input `foobar` the following will write out `foobarbar`:

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

An argument of zero to `yyless` will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using `BEGIN`, for example), this will result in an endless loop.

Note that `yyless` is a macro and can only be used in the `flex` input file, not from other source files.

`unput(c)` puts the character *c* back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

```
{
int i;
/* Copy yytext because unput() trashes yytext */
char *yycopy = strdup( yytext );
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
 unput( yycopy[i] );
 unput( '(' );
 free( yycopy );
}
```

Note that since each `unput()` puts the given character back at the beginning of the input stream, pushing back strings must be done back-to-front.

An important potential problem when using `unput()` is that if you are using `%pointer` (the default), a call to `unput()` *destroys* the contents of `yytext`, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to `unput()` (as in the above example), you must either first copy it elsewhere, or build your scanner using `%array` instead (see How The Input Is Matched).

Finally, note that you cannot put back `EOF` to attempt to mark the input stream with an end-of-file.

`-input()` reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
"/*"        {
register int c;
for ( ; ; )
{
while ( (c = input()) != '*' && c != EOF ) ;
    /* eat up text of comment */
if ( c == '*' )
{
while ( (c = input()) == '*' );
 ( c == '/' )
  break;    /* found the end */
  }
  if ( c == EOF )
  {
  error( "EOF in comment" );
  break;
  }
  }
  }
```

(Note that if the scanner is compiled using `C++`, then `input()` is instead referred to as `yyinput()`, in order to avoid a name clash with the `C++` stream by the name of `input`.)

`YY_FLUSH_BUFFER`
flushes the scanner's internal buffer so that the next time the scanner attempts to match a token, it will first refill the buffer using `YY_INPUT` (see The Generated Scanner, below). This action is a special case of the more general `yy_flush_buffer()` function, described below in the section Multiple Input Buffers.

`yyterminate()` can be used in lieu of a return statement in an action. It terminates the scanner and returns a zero to the scanner's caller, indicating *all done*. By default, `yyterminate()` is also called when an end-of-file is encountered. It is a macro and may be redefined.

THE GENERATED SCANNER

The output of `flex` is the file `lex.yy.c`, which contains the scanning routine `yylex()`, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros. By default, `yylex()` is declared as follows:

```
int yylex()
{
... various definitions and the actions in here ...
}
```

(If your environment supports function prototypes, then it will be `int yylex( void )`.) This definition may be changed by defining the `YY_DECL` macro. For example, you could use:

```
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give the scanning routine the name `lexscan`, returning a float, and taking two floats as arguments. Note that if you give arguments to the scanning routine using a K&R-style/non-prototyped function declaration, you must terminate the definition with a semicolon (;).

Whenever `yylex()` is called, it scans tokens from the global input file `yyin` (which defaults to `stdin`). It continues until it either reaches an end-of-file (at which point it returns the value zero) or one of its actions executes a `return` statement.

If the scanner reaches an end-of-file, subsequent calls are undefined unless either `yyin` is pointed at a new input file (in which case scanning continues from that file), or `yyrestart()` is called. `yyrestart()` takes one argument, a `FILE *` pointer (which can be nil, if you've set up `YY_INPUT` to scan from a source other than `yyin`), and initializes `yyin` for scanning from that file. Essentially there is no difference between just assigning `yyin` to a new input file or using `yyrestart()` to do so; the latter is available for compatibility with previous versions of `flex`, and because it can be used to switch input files in the middle of scanning. It can also be used to throw away the current input buffer, by calling it with an argument of `yyin`; but better is to use `YY_FLUSH_BUFFER` (see above). Note that `yyrestart()` does *not* reset the start condition to `INITIAL` (see Start Conditions, below).

If `yylex()` stops scanning due to executing a `return` statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off.

By default (and for purposes of efficiency), the scanner uses block-reads rather than simple `getc()` calls to read characters from `yyin`. The nature of how it gets its input can be controlled by defining the `YY_INPUT` macro. The call interface for `YY_INPUT` is

`YY_INPUT(`*buf*`,` *result*`,` *max_size*`)`

Its action is to place up to *max_size* characters in the character array *buf* and return in the integer variable *result* either the number of characters read or the constant `YY_NULL` (zero on Unix systems) to indicate EOF. The default `YY_INPUT` reads from the global file-pointer `yyin`.

A sample definition of `YY_INPUT` (in the definitions section of the input file):

```
%{
#define YY_INPUT(buf,result,max_size) \
{ \
  int c = getchar(); \
  result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
}
%}
```

This definition will change the input processing to occur one character at a time.

When the scanner receives an end-of-file indication from YY_INPUT, it then checks the `yywrap()` function. If `yywrap()` returns false (zero), then it is assumed that the function has gone ahead and set up `yyin` to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning zero to its caller. Note that in either case, the start condition remains unchanged; it does *not* revert to `INITIAL`.

If you do not supply your own version of `yywrap()`, then you must either use `%option noyywrap` (in which case the scanner behaves as though `yywrap()` returned one), or you must link with **-lfl** to obtain the default version of the routine, which always returns one.

Three routines are available for scanning from in-memory buffers rather than files: `yy_scan_string()`, `yy_scan_bytes()`, and `yy_scan_buffer()`. See the discussion of them below in the section Multiple Input Buffers.

The scanner writes its `ECHO` output to the `yyout` global (default, `stdout`), which may be redefined by the user simply by assigning it to some other `FILE` pointer.

START CONDITIONS

`flex` provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with <*sc*> will only be active when the scanner is in the start condition named *sc*. For example,

```
<STRING>[^"]*    { /* eat up the string body ... */
...
}
```

will be active only when the scanner is in the `STRING` start condition, and

```
<INITIAL,STRING,QUOTE>\.   { /* handle an escape ... */
...
}
```

will be active only when the current start condition is either `INITIAL`, `STRING`, or `QUOTE`.

Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either `%s` or `%x` followed by a list of names. The former declares *inclusive* start conditions, the latter *exclusive* start conditions. A start condition is activated using the `BEGIN` action. Until the next `BEGIN` action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive. If the start condition is inclusive, then rules with no start conditions at all will also be active. If it is exclusive, then *only* rules qualified with the start condition will be active. A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the flex input. Because of this, exclusive start conditions make it easy to specify mini-scanners which scan portions of the input that are syntactically different from the rest (e.g., comments).

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%
<example>foo   do_something();
bar          something_else();
```

is equivalent to

```
%x example
%%
<example>foo   do_something();
<INITIAL,example>bar    something_else();
```

Without the <INITIAL,example> qualifier, the bar pattern in the second example wouldn't be active (i.e., couldn't match) when in start condition example. If we just used <example> to qualify bar, though, then it would only be active in example and not in INITIAL, while in the first example it's active in both, because in the first example the example start condition is an inclusive (%s) start condition.

Also note that the special start-condition specifier <*> matches every start condition. Thus, the above example could also have been written;

```
%x example
%%
<example>foo   do_something();
<*>bar    something_else();
```

The default rule (to ECHO any unmatched character) remains active in start conditions. It is equivalent to:

```
<*>.|\n     ECHO;
```

BEGIN(0) returns to the original state where only the rules with no start conditions are active. This state can also be referred to as the start-condition INITIAL, so BEGIN(INITIAL) is equivalent to BEGIN(0). (The parentheses around the start condition name are not required but are considered good style.)

BEGIN actions can also be given as indented code at the beginning of the rules section. For example, the following will cause the scanner to enter the SPECIAL start condition whenever yylex() is called and the global variable enter_special is true:

```
int enter_special;
%x SPECIAL
%%
if ( enter_special )
BEGIN(SPECIAL);
<SPECIAL>blahblahblah
...more rules follow...
```

To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like 123.456. By default it will treat it as three tokens, the integer 123, a dot ('.'), and the integer 456. But if the string is preceded earlier in the line by the string expect-floats it will treat it as a single token, the floating-point number 123.456:

```
%{
#include <math.h>
%}
%s expect
%%
expect-floats        BEGIN(expect);
<expect>[0-9]+"."[0-9]+      {
     printf( "found a float, = %f\n",atof( yytext ) );
     }
<expect>\n           {
    /* that's the end of the line, so
   * we need another "expect-number"
   * before we'll recognize any more
   * numbers
   */
   BEGIN(INITIAL);
                 }
[0-9]+      {
     printf( "found an integer, = %d\n",
```

```
     atoi( yytext ) );
     }
"."  printf( "found a dot\n" );
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line.

```
%x comment
%%
int line_num = 1;
"/*"         BEGIN(comment);
<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This scanner goes to a bit of trouble to match as much text as possible with each rule. In general, when attempting to write a high-speed scanner try to match as much possible in each rule, as it's a big win.

Note that start-conditions names are really integer values and can be stored as such. Thus, the above could be extended in the following fashion:

```
%x comment foo
%%
int line_num = 1;
int comment_caller;
"/*"         {
comment_caller = INITIAL;
BEGIN(comment);
}
...
<foo>"/*"    {
comment_caller = foo;
BEGIN(comment);
}
<comment>[^*\n]*       /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

Furthermore, you can access the current start condition using the integer-valued YY_START macro. For example, the above assignments to comment_caller could instead be written

```
comment_caller = YY_START;
```

flex provides YYSTATE as an alias for YY_START (since that is what's used by AT&T lex).

Note that start conditions do not have their own name-space; %s directives and %x directives declare names in the same fashion as #define directives.

Finally, here's an example of how to match C-style quoted strings using exclusive start conditions, including expanded escape sequences (but not including checking for a string that's too long):

```
%x str
%%
char string_buf[MAX_STR_CONST];
char *string_buf_ptr;
\"      string_buf_ptr = string_buf; BEGIN(str);
<str>\"        { /* saw closing quote - all done */
BEGIN(INITIAL);
*string_buf_ptr = '\0';
/* return string constant token type and
* value to parser
*/
}
<str>\n        {
/* error - unterminated string constant */
/* generate error message */
}
<str>\\[0-7]{1,3} {
/* octal escape sequence */
int result;
(void) sscanf( yytext + 1, "%o", &result );
if ( result > 0xff )
/* error, constant is out-of-bounds */
*string_buf_ptr++ = result;
}
<str>\\[0-9]+ {
/* generate error - bad escape sequence; something
* like '\48' or '\0777777' */
```

```
}
<str>\\n  *string_buf_ptr++ = '\n';
<str>\\t  *string_buf_ptr++ = '\t';
<str>\\r  *string_buf_ptr++ = '\r';
<str>\\b  *string_buf_ptr++ = '\b';
<str>\\f  *string_buf_ptr++ = '\f';
<str>\\(.|\n)  *string_buf_ptr++ = yytext[1];
<str>[^\\\n\"]+        {
char *yptr = yytext;
while ( *yptr )
*string_buf_ptr++ = *yptr++;
}
```

Often, such as in some of the examples above, you wind up writing a whole bunch of rules all preceded by the same start condition(s). `flex` makes this a little easier and cleaner by introducing a notion of start condition *scope*. A start condition scope is begun with:

<*SCs*>{

where *SCs* is a list of one or more start conditions. Inside the start condition scope, every rule automatically has the prefix *<SCs>* applied to it, until a '}' which matches the initial '{'. So, for example,

```
<ESC>{
        "\\n"   return '\n';
        "\\r"   return '\r';
        "\\f"   return '\f';
        "\\0"   return '\0';
    }
```

is equivalent to:

```
    <ESC>"\\n"  return '\n';
    <ESC>"\\r"  return '\r';
    <ESC>"\\f"  return '\f';
    <ESC>"\\0"  return '\0';
```

Start condition scopes may be nested.

Three routines are available for manipulating stacks of start conditions:

`void yy_push_state(int` *new_state*`)`
: pushes the current start condition onto the top of the start condition stack and switches to *new_state* as though you had used `BEGIN` *new_state* (recall that start condition names are also integers).

`void yy_pop_state()`
: pops the top of the stack and switches to it via `BEGIN`.

`int yy_top_state()`
: returns the top of the stack without altering the stack's contents.

The start condition stack grows dynamically and so has no built-in size limitation. If memory is exhausted, program execution aborts.

To use start condition stacks, your scanner must include a `%option stack` directive (see Options below).

MULTIPLE INPUT BUFFERS

Some scanners (such as those which support include files) require reading from several input streams. As `flex` scanners do a large amount of buffering, one cannot control where the next input will be read from by simply writing a `YY_INPUT` which is sensitive to the scanning context. `YY_INPUT` is only called when the scanner reaches the end of its buffer, which may be a long time after scanning a statement such as an `include` which requires switching the input source.

To negotiate these sorts of problems, `flex` provides a mechanism for creating and switching between multiple input buffers. An input buffer is created by using `yy_create_buffer()`. This and other related functions follow:

`YY_BUFFER_STATE yy_create_buffer( FILE *`*file*`, int` *size* `)`
: takes a `FILE` pointer and a size and creates a buffer associated with the given file and large enough to hold *size* characters (when in doubt, use `YY_BUF_SIZE` for the size). It returns a `YY_BUFFER_STATE` handle, which may then be passed to other routines (see below). The `YY_BUFFER_STATE` type is a pointer to an opaque `struct yy_buffer_state` structure, so you may safely

initialize `YY_BUFFER_STATE` variables to `((YY_BUFFER_STATE) 0)` if you wish, and also refer to the opaque structure in order to correctly declare input buffers in source files other than that of your scanner. Note that the `FILE` pointer in the call to `yy_create_buffer` is only used as the value of `yyin` seen by `YY_INPUT`; if you redefine `YY_INPUT` so it no longer uses `yyin`, then you can safely pass a nil `FILE` pointer to `yy_create_buffer`. You select a particular buffer to scan from using:

`void yy_switch_to_buffer( YY_BUFFER_STATE` *new_buffer* `)`

switches the scanner's input buffer so subsequent tokens will come from *new_buffer*. Note that `yy_switch_to_buffer()` may be used by `yywrap()` to set things up for continued scanning, instead of opening a new file and pointing `yyin` at it. Note also that switching input sources via either `yy_switch_to_buffer()` or `yywrap()` does *not* change the start condition.

`void yy_delete_buffer( YY_BUFFER_STATE` *buffer* `)`

is used to reclaim the storage associated with a buffer. (buffer can be nil, in which case the routine does nothing.)

You can also clear the current contents of a buffer using the `yy_flush_buffer()` function.

`void yy_flush_buffer( YY_BUFFER_STATE` *buffer* `)`

This function discards the buffer's contents, so the next time the scanner attempts to match a token from the buffer, it will first fill the buffer anew using `YY_INPUT`.

`yy_new_buffer()`

is an alias for `yy_create_buffer()`, provided for compatibility with the C++ use of `new` and `delete` for creating and destroying dynamic objects.

`YY_CURRENT_BUFFER()`

is a macro that returns a `YY_BUFFER_STATE` handle to the current buffer.

Here is an example of using these facilities for writing a scanner which expands include files (the `<<EOF>>` feature is discussed below):

```
/* the "incl" state is used for picking up the name
* of an include file
*/
%x incl
%{
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_ptr = 0;
%}
%%
include             BEGIN(incl);
[a-z]+              ECHO;
[^a-z\n]*\n?        ECHO;
<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+   { /* got the include file name */
if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )
{
fprintf( stderr, "Includes nested too deeply" );
exit( 1 );
}
include_stack[include_stack_ptr++] =
YY_CURRENT_BUFFER;
yyin = fopen( yytext, "r" );
if ( ! yyin )
error( ... );
yy_switch_to_buffer(
yy_create_buffer( yyin, YY_BUF_SIZE ) );
BEGIN(INITIAL);
}
<<EOF>> {
if ( --include_stack_ptr < 0 )
{
yyterminate();
} else
{
yy_delete_buffer( YY_CURRENT_BUFFER );
yy_switch_to_buffer(
include_stack[include_stack_ptr] );
}
}
```

Three routines are available for setting up input buffers for scanning in-memory strings instead of files. All of them create a new input buffer for scanning the string, and return a corresponding `YY_BUFFER_STATE` handle (which you should delete with `yy_delete_buffer()` when done with it). They also switch to the new buffer using `yy_switch_to_buffer()`, so the next call to `yylex()` will start scanning the string.

`yy_scan_string(const char *`*str*`)`
: scans a NUL-terminated string.

`yy_scan_bytes(const char *`*bytes*`, int` *len*`)`
: scans *len* bytes (including possibly NUL values) starting at location *bytes.*

Note that both of these functions create and scan a copy of the string or bytes. (This may be desirable, since `yylex()` modifies the contents of the buffer it is scanning.) You can avoid the copy by using:

`yy_scan_buffer(char *`*base*`, yy_size_t` *size*`)`
: which scans in place the buffer starting at *base,* consisting of *size* bytes, the last two bytes of which *must* be `YY_END_OF_BUFFER_CHAR` (ASCII `NUL`). These last two bytes are not scanned; thus, scanning consists of `base[0]` through `base[`*size*`-2]`, inclusive.

  If you fail to set up *base* in this manner (i.e., forget the final two `YY_END_OF_BUFFER_CHAR` bytes), then `yy_scan_buffer()` returns a nil pointer instead of creating a new input buffer.

  The type `yy_size_t` is an integral type to which you can cast an integer expression reflecting the size of the buffer.

END-OF-FILE RULES

The special rule `<<EOF>>` indicates actions which are to be taken when an end-of-file is encountered and yywrap() returns non-zero (i.e., indicates no further files to process). The action must finish by doing one of four things:

- assigning `yyin` to a new input file (in previous versions of `flex`, after doing the assignment you had to call the special action `YY_NEW_FILE`; this is no longer necessary);

- executing a `return` statement;
- executing the special `yyterminate()` action;
- switching to a new buffer using `yy_switch_to_buffer()` as shown in the example above.

`<<EOF>>` rules may not be used with other patterns; they may only be qualified with a list of start conditions. If an unqualified `<<EOF>>` rule is given, it applies to *all* start conditions which do not already have `<<EOF>>` actions. To specify an `<<EOF>>` rule for only the initial start condition, use

```
<INITIAL><<EOF>>
```

These rules are useful for catching things like unclosed comments. An example:

```
%x quote
%%
```

...other rules for dealing with quotes...

```
<quote><<EOF>>   {
error( "unterminated quote" );
yyterminate();
}
<<EOF>>  {
if ( *++filelist )
yyin = fopen( *filelist, "r" );
else
yyterminate();
}
```

MISCELLANEOUS MACROS

The macro `YY_USER_ACTION` can be defined to provide an action which is always executed prior to the matched rule's action. For example, it could be reset using a `#define` directive to call a routine to convert `yytext` to lower-case. When `YY_USER_ACTION` is invoked, the variable `yy_act` gives the number of the matched rule (rules are numbered starting with one). Suppose you want to profile how often each of your rules is matched. The following would do the trick:

```
#define YY_USER_ACTION ++ctr[yy_act]
```

where `ctr` is an array to hold the counts for the different rules. Note that the macro `YY_NUM_RULES` gives the total number of rules (including the default rule, even if you use **-s**), so a correct declaration for `ctr` is:

```
int ctr[YY_NUM_RULES];
```

The macro `YY_USER_INIT` may be defined to provide an action which is always executed before the first scan (and before the scanner's internal initializations are done). For example, it could be used to call a routine to read in a data table or open a logging file.

The macro `yy_set_interactive(is_interactive)` can be used to control whether the current buffer is considered interactive. An interactive buffer is processed more slowly, but must be used when the scanner's input source is indeed interactive to avoid problems due to waiting to fill buffers (see the discussion of the **-I** flag below). A non-zero value in the macro invocation marks the buffer as interactive, a zero value as non-interactive. Note that use of this macro overrides `%option always-interactive` or `%option never-interactive` (see Options below). `yy_set_interactive()` must be invoked prior to beginning to scan the buffer that is (or is not) to be considered interactive.

The macro `yy_set_bol(at_bol)` can be used to control whether the current buffer's scanning context for the next token match is done as though at the beginning of a line. A non-zero macro argument makes rules anchored with a '^' character active, while a zero argument make rules anchored with a '^' character inactive.

The macro `YY_AT_BOL()` returns true if the next token scanned from the current buffer will have '^' rules active, false otherwise.

In the generated scanner, the actions are all gathered in one large `switch` statement and separated using `YY_BREAK`, which may be redefined. By default, it is simply a `break`, to separate each rule's action from the following rule's action. By redefining `YY_BREAK` with a `#define` directive so that it does nothing (while being very careful that every rule ends with a `break` or a `return`), C++ users can avoid suffering from unreachable statement warnings where because a rule's action ends with `return`, the `YY_BREAK` is inaccessible.

VALUES AVAILABLE TO THE USER

This section summarizes the various values available to the user in the rule actions.

`char *yytext` holds the text of the current token. It may be modified but not lengthened (you cannot append characters to the end).

If the special directive `%array` appears in the first section of the scanner description, then `yytext` is instead declared `char yytext[YYLMAX]`, where `YYLMAX` is a macro definition that you can redefine in the first section if you don't like the default value (generally 8KB). Using `%array` results in somewhat slower scanners, but the value of `yytext` becomes immune to calls to `input()` and `unput()`, which potentially destroy its value when `yytext` is a character pointer. The opposite of `%array` is `%pointer`, which is the default.

You cannot use `%array` when generating C++ scanner classes (the **-+** flag).

`int yyleng` holds the length of the current token.

`FILE *yyin` is the file which by default `flex` reads from. It may be redefined but doing so only makes sense before scanning begins or after an EOF has been encountered. Changing it in the midst of scanning will have unexpected results since `flex` buffers its input; use `yyrestart()` instead. Once scanning terminates because an end-of-file has been seen, you can assign `yyin` at the new input file and then call the scanner again to continue scanning.

`void yyrestart( FILE *`*new_file* `)`
may be called to point `yyin` at the new input file. The switch-over to the new file is immediate (any previously buffered-up input is lost). Note that calling `yyrestart()` with `yyin` as an argument thus throws away the current input buffer and continues scanning the same input file.

`FILE *yyout` is the file to which `ECHO` actions are done. It can be reassigned by the user.

`YY_CURRENT_BUFFER`
returns a `YY_BUFFER_STATE` handle to the current buffer.

`YY_START` returns an integer value corresponding to the current start condition. You can subsequently use this value with `BEGIN` to return to that start condition.

INTERFACING WITH YACC

One of the main uses of `flex` is as a companion to the `yacc` parser-generator. `yacc` parsers expect to call a routine named `yylex()` to find the next input token. The routine is supposed to return the type of the next token as well as putting any associated value in the global `yylval`. To use `flex` with `yacc`, one specifies the **-d** option to `yacc` to instruct it to generate the file `y.tab.h` containing definitions of all the `%tokens` appearing in the `yacc` input. This file is then included in the `flex` scanner. For example, if one of the tokens is `TOK_NUMBER`, part of the scanner might look like:

```
%{
#include "y.tab.h"
%}
%%
[0-9]+     yylval = atoi( yytext ); return TOK_NUMBER;
```

OPTIONS

`flex` has the following options:

-b Generate backing-up information to `lex.backup`. This is a list of scanner states which require backing up and the input characters on which they do so. By adding rules one can remove backing-up states. If *all* backing-up states are eliminated and **-Cf** or **-CF** is used, the generated scanner will run faster (see the **-p** flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option. (See the section on Performance Considerations below.)

–c is a do-nothing, deprecated option included for POSIX compliance.

–d makes the generated scanner run in `debug` mode. Whenever a pattern is recognized and the global `yy_flex_debug` is non-zero (which is the default), the scanner will write to `stderr` a line of the form:

`--accepting rule at line 53 ("`*matched text*`")`

The line number refers to the location of the rule in the file defining the scanner (i.e., the file that was fed to `flex`). Messages are also generated when the scanner backs up, accepts the default rule, reaches the end of its input buffer (or encounters a NUL; at this point, the two look the same as far as the scanner's concerned), or reaches an end-of-file.

–f specifies *fast scanner*. No table compression is done and stdio is bypassed. The result is large but fast. This option is equivalent to **–Cfr** (see below).

–h generates a *help* summary of options to `stdout` and then exits. **–?** and **--help** are synonyms for **–h.**

–i instructs `flex` to generate a *case-insensitive* scanner. The case of letters given in the `flex` input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in `yytext` will have the preserved case (i.e., it will not be folded).

–l turns on maximum compatibility with the original AT&T `lex` implementation. Note that this does not mean *full* compatibility. Use of this option costs a considerable amount of performance, and it cannot be used with the **–+**, **–f, –F, –Cf,** or **–CF** options. For details on the compatibilities it provides, see the section "Incompatibilities With Lex And POSIX" below. This option also results in the name `YY_FLEX_LEX_COMPAT` being set with a `#define` directive in the generated scanner.

–n is another do-nothing, deprecated option included only for POSIX compliance.

-p generates a performance report to `stderr`. The report consists of comments regarding features of the `flex` input file which will cause a serious loss of performance in the resulting scanner. If you give the flag twice, you will also get comments regarding features that lead to minor performance losses.

Note that the use of `REJECT`, `%option yylineno`, and variable trailing context (see the Deficiencies/Bugs section below) entails a substantial performance penalty; use of `yymore()`, the ^ operator, and the **-I** flag entail minor performance penalties.

-s causes the default rule (that unmatched scanner input is echoed to `stdout`) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

-t instructs `flex` to write the scanner it generates to standard output instead of `lex.yy.c`.

-v specifies that `flex` should write to `stderr` a summary of statistics regarding the scanner it generates. Most of the statistics are meaningless to the casual `flex` user, but the first line identifies the version of `flex` (same as reported by **-V)**, and the next line the flags used when generating the scanner, including those that are on by default.

-w suppresses warning messages.

-B instructs `flex` to generate a *batch* scanner, the opposite of interactive scanners generated by **-I** (see below). In general, you use **-B** when you are certain that your scanner will never be used interactively, and you want to squeeze a little more performance out of it. If your goal is instead to squeeze out a significantly more performance, you should be using the **-Cf** or **-CF** options (discussed below), which turn on **-B** automatically anyway.

-F specifies that the *fast* scanner table representation should be used (and stdio bypassed). This representation is about as fast as the full table representation (**-f**), and for some sets of patterns will be considerably smaller (and for others, larger). In general, if the pattern set contains both keywords and a catch-all, identifier rule, such as in the set:

```
"case"    return TOK_CASE;
"switch"  return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+    return TOK_ID;
```

then you're better off using the full table representation. If only the identifier rule is present and you then use a hash table or some such to detect the keywords, you're better off using **-F**.

This option is equivalent to **-CFr** (see below). It cannot be used with **-+**.

-I instructs flex to generate an *interactive* scanner. An interactive scanner is one that only looks ahead to decide what token has been matched if it absolutely must. It turns out that always looking one extra character ahead, even if the scanner has already seen enough text to disambiguate the current token, is a bit faster than only looking ahead when necessary. But scanners that always look ahead give dreadful interactive performance; for example, when a user types a newline, it is not recognized as a newline token until they enter another token, which often means typing in another whole line.

flex scanners default to interactive unless you use the **-Cf** or **-CF** table-compression options (see below). That's because if you're looking for high-performance you should be using one of these options, so if you didn't, flex assumes you'd rather trade off a bit of run-time performance for intuitive interactive behavior. Note also that you *cannot* use **-I** in conjunction with **-Cf** or **-CF**. Thus, this option is not really needed; it is on by default for all those cases in which it is allowed.

You can force a scanner to *not* be interactive by using **-B** (see above).

-L instructs `flex` not to generate `#line` directives. Without this option, `flex` peppers the generated scanner with `#line` directives so error messages in the actions will be correctly located with respect to either the original `flex` input file (if the errors are due to code in the input file), or `lex.yy.c` (if the errors are the fault of `flex`, you should report these sorts of errors to the email address given below).

-T makes `flex` run in *trace* mode. It will generate a lot of messages to `stderr` concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option is mostly for use in maintaining `flex`.

-V prints the version number to `stdout` and exits. **--version** is a synonym for **-V.**

-7 instructs `flex` to generate a 7-bit scanner, i.e., one which can only recognized 7-bit characters in its input. The advantage of using **-7** is that the scanner's tables can be up to half the size of those generated using the **-8** option (see below). The disadvantage is that such scanners often hang or crash if their input contains an 8-bit character.

Note, however, that unless you generate your scanner using the **-Cf** or **-CF** table compression options, use of **-7** will save only a small amount of table space, and make your scanner considerably less portable. The default behavior of `flex` is to generate an 8-bit scanner unless you use the **-Cf** or **-CF,** in which case `flex` defaults to generating 7-bit scanners unless your site was always configured to generate 8-bit scanners (as will often be the case with non-USA sites). You can tell whether `flex` generated a 7-bit or an 8-bit scanner by inspecting the flag summary in the **-v** output as described above.

Note that if you use **-Cfe** or **-CFe** (those table compression options, but also using equivalence classes as discussed see below), `flex` still defaults to generating an 8-bit scanner, since usually with these compression options full 8-bit tables are not much more expensive than 7-bit tables.

-8 instructs `flex` to generate an 8-bit scanner, i.e., one which can recognize 8-bit characters. This flag is only needed for scanners generated using **-Cf** or **-CF,** as otherwise `flex` defaults to generating an 8-bit scanner anyway.

See the discussion of **-7** above for the default behavior and the tradeoffs between 7-bit and 8-bit scanners.

-+ specifies that you want `flex` to generate a C++ scanner class. See the section on Generating C++ Scanners below for details.

-C[aefFmr] controls the degree of table compression and, more generally, trade-offs between small scanners and fast scanners.

-Ca (*align*) instructs `flex` to trade off larger tables in the generated scanner for faster performance because the elements of the tables are better aligned for memory access and computation. On some RISC architectures, fetching and manipulating longwords is more efficient than with smaller-sized units such as shortwords. This option can double the size of the tables used by your scanner.

-Ce directs `flex` to construct *equivalence classes*, i.e., sets of characters which have identical lexical properties (for example, if the only appearance of digits in the `flex` input is in the character class `[0-9]` then the digits '`0`', '`1`', ..., '`9`' will all be put in the same equivalence class). Equivalence classes usually give dramatic reductions in the final table/object file sizes (typically a factor of 2-5) and are pretty cheap performance-wise (one array look-up per character scanned).

-Cf specifies that the *full* scanner tables should be generated – `flex` should not compress the tables by taking advantages of similar transition functions for different states.

-CF specifies that the alternate *fast* scanner representation (described above under the **-F** flag) should be used. This option cannot be used with **-+**.

-Cm directs `flex` to construct *meta-equivalence classes*, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. Meta-equivalence classes are often a big win when using compressed tables, but they have a moderate performance impact (one or two `if` tests and one array look-up per character scanned).

-Cr causes the generated scanner to bypass use of the standard I/O library (`stdio`) for input. Instead of calling `fread()` or `getc()`, the scanner will use the `read()` system call, resulting in a performance gain which varies from system to system, but in general is probably negligible unless you are also using **-Cf** or **-CF**. Using **-Cr** can cause strange behavior if, for example, you read from `yyin` using `stdio` prior to calling the scanner (because the scanner will miss whatever text your previous reads left in the `stdio` input buffer).

-Cr has no effect if you define `YY_INPUT` (see The Generated Scanner above).

A lone **-C** specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

The options **–Cf** or **–CF** and **–Cm** do not make sense together – there is no opportunity for meta-equivalence classes if the table is not being compressed. Otherwise the options may be freely mixed, and are cumulative.

The default setting is **–Cem**, which specifies that `flex` should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. You can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

slowest & smallest
–Cem
–Cm
–Ce
–C
–C{f,F}e
–C{f,F}
–C{f,F}a
fastest & largest

Note that scanners with the smallest tables are usually generated and compiled the quickest, so during development you will usually want to use the default, maximal compression.

–Cfe is often a good compromise between speed and size for production scanners.

–o*output* directs `flex` to write the scanner to the file `output` instead of `lex.yy.c`. If you combine **–o** with the **–t** option, then the scanner is written to `stdout` but its `#line` directives (see the **–L** option above) refer to the file `output`.

–P*prefix* changes the default `yy` prefix used by `flex` for all globally-visible variable and function names to instead be *prefix*. For example, **–P**`foo` changes the name of yytext to footext. It also changes the name of the default output file from lex.yy.c to lex.foo.c. Here are all of the names affected:

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

(If you are using a C++ scanner, then only yywrap and yyFlexLexer are affected.) Within your scanner itself, you can still refer to the global variables and functions using either version of their name; but externally, they have the modified name.

This option lets you easily link together multiple flex programs into the same executable. Note, though, that using this option also renames yywrap(), so you now *must* either provide your own (appropriately-named) version of the routine for your scanner, or use %option noyywrap, as linking with **-lfl** no longer provides one for you by default.

-S*skeleton_file* overrides the default skeleton file from which flex constructs its scanners. You'll never need this option unless you are doing flex maintenance or development.

flex also provides a mechanism for controlling options within the scanner specification itself, rather than from the flex command-line. This is done by including %option directives in the first section of the scanner specification. You can specify multiple options with a single %option directive, and multiple directives in the first section of your flex input file.

Most options are given simply as names, optionally preceded by the word `no` (with no intervening whitespace) to negate their meaning. A number are equivalent to `flex` flags or their negation:

`7bit`	**-7** option
`8bit`	**-8** option
`align`	-**Ca** option
`backup`	-**b** option
`batch`	-**B** option
`c++`	**-+** option
`caseful`, `case-sensitive`	opposite of**-i** (default)
`case-insensitive`, `caseless`	**-i** option
`debug`	-**d** option
`default`	opposite of**-s** option
`ecs`	**-Ce** option
`fast`	**-F** option
`full`	**-f** option
`interactive`	**-I** option
`lex-compat`	-**l** option
`meta-ecs`	**-Cm** option
`perf-report`	**-p** option
`read`	**-Cr** option
`stdout`	-**t** option
`verbose`	**-v** option
`warn`	opposite of**-w** option (use `%option nowarn` to invoke **-w**)
`array`	equivalent to `%array`

`pointer`	equivalent to `%pointer` (default)

Some `%` options provide features otherwise not available:

`always-interactive`
: instructs `flex` to generate a scanner which always considers its input interactive. Normally, on each new input file the scanner calls `isatty()` in an attempt to determine whether the scanner's input source is interactive and thus should be read a character at a time. When this option is used, however, then no such call is made.

`main`
: directs `flex` to provide a default `main()` program for the scanner, which simply calls `yylex()`. This option implies `noyywrap` (see below).

`never-interactive`
: instructs `flex` to generate a scanner which never considers its input interactive (again, no call made to `isatty()`). This is the opposite of `always-interactive`.

`stack`
: enables the use of start condition stacks (see Start Conditions above).

`stdinit`
: if set (i.e., `%option stdinit)` initializes `yyin` and `yyout` to `stdin` and `stdout`, instead of the default of `nil`. Some existing `lex` programs depend on this behavior, even though it is not compliant with ANSI C, which does not require `stdin` and `stdout` to be compile-time constant.

`yylineno`
: directs `flex` to generate a scanner that maintains the number of the current line read from its input in the global variable `yylineno`. This option is implied by `%option lex-compat`.

`yywrap`
: if unset (i.e., `%option noyywrap)`, makes the scanner not call `yywrap()` upon an end-of-file, but simply assume that there are no more files to scan (until the user points `yyin` at a new file and calls `yylex()` again).

flex scans your rule actions to determine whether you use the REJECT or yymore() features. The reject and yymore options are available to override its decision as to whether you use the options, either by setting them (e.g., %option reject) to indicate the feature is indeed used, or unsetting them to indicate it actually is not used (e.g., %option noyymore).

Three options take string-delimited values, offset with '=':

%option outfile="ABC"
is equivalent to **-o**ABC

%option prefix="XYZ"
is equivalent to **-P**XYZ.

%option yyclass="foo"
only applies when generating a C++ scanner (**-+** option). It informs flex that you have derived foo as a subclass of yyFlexLexer, so flex will place your actions in the member function foo::yylex() instead of yyFlexLexer::yylex(). It also generates a yyFlexLexer::yylex() member function that emits a run-time error (by invoking yyFlexLexer::LexerError()) if called. See Generating C++ Scanners, below, for additional information.

A number of options are available for lint purists who want to suppress the appearance of unneeded routines in the generated scanner. Each of the following, if unset (e.g., %option nounput), results in the corresponding routine not appearing in the generated scanner:

```
input
unput
yy_push_state
yy_pop_state
yy_top_state
yy_scan_buffer
yy_scan_bytes
yy_scan_string
```

(yy_push_state() and friends won't appear anyway unless you use %option stack).

PERFORMANCE CONSIDERATIONS

The main design goal of `flex` is that it generate high-performance scanners. It has been optimized for dealing well with large sets of rules. Aside from the effects on scanner speed of the table compression **-C** options outlined above, there are a number of options/actions which degrade performance. These are, from most expensive to least:

`REJECT`

`%option yylineno`

`arbitrary trailing context`

pattern sets (those that require backing up)

`%array`

`%option interactive`

`%option always-interactive`

`^` (the beginning-of-line operator)

`yymore()`

with the first three all being quite expensive and the last two being quite cheap. Note also that `unput()` is implemented as a routine call that potentially does quite a bit of work, while `yyless()` is a quite-cheap macro; so if just putting back some excess text you scanned, use `yyless()`.

`REJECT` should be avoided at all costs when performance is important. It is a particularly expensive option.

Getting rid of backing up is messy and often may be an enormous amount of work for a complicated scanner. In principal, one begins by using the **-b** flag to generate a `lex.backup` file. For example, on the input

```
%%
foo        return TOK_KEYWORD;
foobar     return TOK_KEYWORD;
```

the file looks like:

```
State #6 is non-accepting -
associated rule line numbers:
2 3
out-transitions: [ o ]
jam-transitions: EOF [ \001-n  p-\177 ]
State #8 is non-accepting -
associated rule line numbers:
3
out-transitions: [ a ]
jam-transitions: EOF [ \001-`  b-\177 ]
State #9 is non-accepting -
associated rule line numbers:
3
out-transitions: [ r ]
jam-transitions: EOF [ \001-q  s-\177 ]
Compressed tables always back up.
```

The first few lines tell us that there's a scanner state in which it can make a transition on an '`o`' but not on any other character, and that in that state the currently scanned text does not match any rule. The state occurs when trying to match the rules found at lines 2 and 3 in the input file. If the scanner is in that state and then reads something other than an '`o`', it will have to back up to find a rule which is matched. With a bit of headscratching one can see that this must be the state it's in when it has seen `fo`. When this has happened, if anything other than another '`o`' is seen, the scanner will have to back up to simply match the '`f`' (by the default rule).

The comment regarding State #8 indicates there's a problem when `foob` has been scanned. Indeed, on any character other than an '`a`', the scanner will have to back up to accept `foo`. Similarly, the comment for State #9 concerns when `fooba` has been scanned and an '`r`' does not follow.

The final comment reminds us that there's no point going to all the trouble of removing backing up from the rules unless we're using **`-Cf`** or **`-CF`,** since there's no performance gain doing so with compressed scanners.

The way to remove the backing up is to add error rules:

```
%%
  foo         return TOK_KEYWORD;
  foobar      return TOK_KEYWORD;
  fooba       |
  foob        |
  fo          {
              /* false alarm, not really a keyword */
              return TOK_ID;
              }
```

Eliminating backing up among a list of keywords can also be done using a *catch-all* rule:

```
%%
  foo         return TOK_KEYWORD;
  foobar      return TOK_KEYWORD;
  [a-z]+      return TOK_ID;
```

This is usually the best solution when appropriate.

Backing up messages tend to cascade. With a complicated set of rules it's not uncommon to get hundreds of messages. If one can decipher them, though, it often only takes a dozen or so rules to eliminate the backing up (though it's easy to make a mistake and have an error rule accidentally match a valid token. A possible future `flex` feature will be to automatically add rules to eliminate backing up).

It's important to keep in mind that you gain the benefits of eliminating backing up only if you eliminate *every* instance of backing up. Leaving just one means you gain nothing.

Variable trailing context (where both the leading and trailing parts do not have a fixed length) entails almost the same performance loss as `REJECT` (i.e., substantial). So when possible a rule like:

```
%%
mouse|rat/(cat|dog)   run();
```

is better written:

```
%%
mouse/cat|dog           run();
rat/cat|dog             run();
```

or as

```
%%
mouse|rat/cat           run();
mouse|rat/dog           run();
```

Note that here the special '|' action does *not* provide any savings, and can even make things worse (see Deficiencies/Bugs below).

Another area where the user can increase a scanner's performance (and one that's easier to implement) arises from the fact that the longer the tokens matched, the faster the scanner will run. This is because with long tokens the processing of most input characters takes place in the (short) inner scanning loop, and does not often have to go through the additional work of setting up the scanning environment (e.g., `yytext)` for the action. Recall the scanner for C comments:

```
%x comment
%%
        int line_num = 1;
"/*"         BEGIN(comment);
<comment>[^*\n]*
<comment>"*"+[^*/\n]*
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

This could be sped up by writing it as:

```
%x comment
%%
        int line_num = 1;
"/*"         BEGIN(comment);
<comment>[^*\n]*
<comment>[^*\n]*\n      ++line_num;
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Now instead of each newline requiring the processing of another action, recognizing the newlines is *distributed* over the other rules to keep the matched text as long as possible. Note that *adding* rules does *not* slow down the scanner! The speed of the scanner is independent of the number of rules or (modulo the considerations given at the beginning of this section) how complicated the rules are with regard to operators such as '*' and '|'.

A final example in speeding up a scanner: suppose you want to scan through a file containing identifiers and keywords, one per line and with no other extraneous characters, and recognize all the keywords. A natural first approach is:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */
.|\n     /* it's not a keyword */
```

To eliminate the back-tracking, introduce a catch-all rule:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* it's a keyword */
[a-z]+   |
.|\n     /* it's not a keyword */
```

Now, if it's guaranteed that there's exactly one word per line, then we can reduce the total number of matches by a half by merging in the recognition of newlines with that of the other tokens:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
```

```
volatile\n |
while\n  /* it's a keyword */
[a-z]+\n |
.|\n     /* it's not a keyword */
```

One has to be careful here, as we have now reintroduced backing up into the scanner. In particular, while we know that there will never be any characters in the input stream other than letters or newlines, `flex` can't figure this out, and it will plan for possibly needing to back up when it has scanned a token like `auto` and then the next character is something other than a newline or a letter. Previously it would then just match the `auto` rule and be done, but now it has no `auto` rule, only an `auto\n` rule. To eliminate the possibility of backing up, we could either duplicate all rules but without final newlines, or, since we never expect to encounter such an input and therefore don't how it's classified, we can introduce one more catch-all rule, this one which doesn't include a newline:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* it's a keyword */
[a-z]+\n |
[a-z]+   |
.|\n     /* it's not a keyword */
```

Compiled with **`-Cf`**, this is about as fast as one can get a `flex` scanner to go for this particular problem.

A final note: `flex` is slow when matching NUL values, particularly when a token contains multiple NUL values. It's best to write rules which match short amounts of text if it's anticipated that the text will often include NUL values.

Another final note regarding performance: as mentioned above in the section How the Input is Matched, dynamically resizing `yytext` to accommodate huge tokens is a slow process because it presently requires that the (huge) token be rescanned from the beginning. Thus if performance is vital, you should attempt to match large quantities of text but not huge quantities, where the cutoff between the two is at about 8K characters/token.

GENERATING C++ SCANNERS

`flex` provides two different ways to generate scanners for use with C++. The first way is to simply compile a scanner generated by `flex` using a C++ compiler instead of a C compiler. You should not encounter any compilations errors (please report any you find to the email address given in the Author section below). You can then use C++ code in your rule actions instead of C code. Note that the default input source for your scanner remains `yyin`, and default echoing is still done to `yyout`. Both of these remain `FILE *` variables and not C++ streams.

You can also use `flex` to generate a C++ scanner class, using the **-+** option (or, equivalently, `%option c++`), which is automatically specified if the name of the `flex` executable ends in a '+', such as `flex++`. When using this option, `flex` defaults to generating the scanner to the file `lex.yy.cc` instead of `lex.yy.c`. The generated scanner includes the header file `FlexLexer.h`, which defines the interface to two C++ classes.

The first class, `FlexLexer`, provides an abstract base class defining the general scanner class interface. It provides the following member functions:

`const char* YYText()`
: returns the text of the most recently matched token, the equivalent of `yytext`.

`int YYLeng()`
: returns the length of the most recently matched token, the equivalent of `yyleng`.

`int lineno() const`
: returns the current input line number (see `%option yylineno`), or `1` if `%option yylineno` was not used.

`void set_debug( int` *flag* `)`
: sets the debugging flag for the scanner, equivalent to assigning to `yy_flex_debug` (see the Options section above). Note that you must build the scanner using `%option debug` to include debugging information in it.

`int debug() const`
: returns the current setting of the debugging flag.

Also provided are member functions equivalent to `yy_switch_to_buffer()`, `yy_create_buffer()` (though the first argument is an `istream*` object pointer and not a `FILE*`), `yy_flush_buffer()`, `yy_delete_buffer()`, and `yyrestart()` (again, the first argument is a `istream*` object pointer).

The second class defined in `FlexLexer.h` is `yyFlexLexer`, which is derived from `FlexLexer`. It defines the following additional member functions:

`yyFlexLexer( istream* arg_yyin=0, ostream* arg_yyout=0 )`
: constructs a `yyFlexLexer` object using the given streams for input and output. If not specified, the streams default to `cin` and `cout`, respectively.

`virtual int yylex()`
: performs the same role is `yylex()` does for ordinary `flex` scanners: it scans the input stream, consuming tokens, until a rule's action returns a value. If you derive a subclass `S` from `yyFlexLexer` and want to access the member functions and variables of `S` inside `yylex()`, then you need to use `%option yyclass="S"` to inform `flex` that you will be using that subclass instead of `yyFlexLexer`. In this case, rather than generating `yyFlexLexer::yylex()`, `flex` generates `S::yylex()` (and also generates a dummy `yyFlexLexer::yylex()` that calls `yyFlexLexer::LexerError()` if called).

`virtual void switch_streams(istream*` *new_in*`=0, ostream*` *new_out*`=0)`
: reassigns `yyin` to *new_in* (if non-nil) and `yyout` to *new_out* (ditto), deleting the previous input buffer if `yyin` is reassigned.

`int yylex( istream*` *new_in*`, ostream*` *new_out*`=0 )`
: first switches the input streams via `switch_streams(`*new_in*`,` *new_out*`)` and then returns the value of `yylex()`.

In addition, `yyFlexLexer` defines the following protected virtual functions which you can redefine in derived classes to tailor the scanner:

`virtual int LexerInput( char* buf, int max_size )`
reads up to *max_size* characters into *buf* and returns the number of characters read. To indicate end-of-input, it returns zero characters. Note that interactive scanners (see the **-B** and **-I** flags) define the macro `YY_INTERACTIVE`. If you redefine `LexerInput()` and need to take different actions depending on whether or not the scanner might be scanning an interactive input source, you can test for the presence of this name via `#ifdef`.

`virtual void LexerOutput( const char* buf, int size )`
writes out *size* characters from the buffer *buf*, which, while NUL-terminated, may also contain internal NUL values if the scanner's rules can match text with NUL values in them.

`virtual void LexerError( const char* msg )`
reports a fatal error message. The default version of this function writes the message to the stream `cerr` and exits.

Note that a `yyFlexLexer` object contains its entire scanning state. Thus you can use such objects to create reentrant scanners. You can instantiate multiple instances of the same `yyFlexLexer` class, and you can also combine multiple C++ scanner classes together in the same program using the **-P** option discussed above.

Finally, note that the `%array` feature is not available to C++ scanner classes; you must use `%pointer` (the default).

Here is an example of a simple C++ scanner:

```
// An example of using the flex C++ scanner class.
%{
int mylineno = 0;
%}
string  \"[^\n"]+\"
ws      [ \t]+
alpha   [A-Za-z]
dig     [0-9]
name    ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1    [-+]?{dig}+\.?([eE][-+]?{dig}+)?
```

```
num2    [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number  {num1}|{num2}
%%
{ws}    /* skip blanks and tabs */
"/*"    {
int c;
while((c = yyinput()) != 0)
{
if(c == '\n')
++mylineno;
else if(c == '*')
{
if((c = yyinput()) == '/')
break;
else
unput(c);
}
}
}
{number}  cout << "number " << YYText() << '\n';
\n        mylineno++;
{name}    cout << "name " << YYText() << '\n';
{string}  cout << "string " << YYText() << '\n';
%%
int main( int /* argc */, char** /* argv */ )
{
FlexLexer* lexer = new yyFlexLexer;
while(lexer->yylex() != 0)
;
return 0;
}
```

If you want to create multiple (different) lexer classes, you use the **-P** flag (or the `prefix=` *string*) to rename each `yyFlexLexer` to *string*`FlexLexer`. You then can include `<FlexLexer.h>` in your other sources once per lexer class, first renaming `yyFlexLexer` as follows:

```
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>
#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>
```

If, for example, you used `%option prefix="xx"` for one of your scanners and `%option prefix="zz"` for the other.

IMPORTANT: the present form of the scanning class is experimental and may change considerably between major releases.

INCOMPATIBILITIES WITH LEX AND POSIX

`flex` is a rewrite of the AT&T Unix `lex` tool (the two implementations do not share any code, though), with some extensions and incompatibilities, both of which are of concern to those who wish to write scanners acceptable to either implementation. `flex` is fully compliant with the POSIX `lex` specification, except that when using `%pointer` (the default), a call to `unput()` destroys the contents of `yytext`, which is counter to the POSIX specification.

In this section we discuss all of the known areas of incompatibility between `flex`, AT&T `lex`, and the POSIX specification.

The **-l** option turns on maximum compatibility with the original AT&T `lex` implementation, at the cost of a major loss in the generated scanner's performance. We note below which incompatibilities can be overcome using the **-l** option.

`flex` is fully compatible with `lex` with the following exceptions:

- The undocumented `lex` scanner internal variable `yylineno` is not supported unless **-l** or `%option yylineno` is used.

  `yylineno` should be maintained on a per-buffer basis, rather than a per-scanner (single global variable) basis.

  `yylineno` is not part of the POSIX specification.

- The `input()` routine is not redefinable, though it may be called to read characters following whatever has been matched by a rule. If `input()` encounters an end-of-file the normal `yywrap()` processing is done. A real end-of-file is returned by `input()` as EOF.

  Input is instead controlled by defining the `YY_INPUT` macro.

The `flex` restriction that `input()` cannot be redefined is in accordance with the POSIX specification, which simply does not specify any way of controlling the scanner's input other than by making an initial assignment to `yyin`.

- The `unput()` routine is not redefinable. This restriction is in accordance with POSIX.
- `flex` scanners are not as reentrant as `lex` scanners. In particular, if you have an interactive scanner and an interrupt handler which long-jumps out of the scanner, and the scanner is subsequently called again, you may get the following message:

```
fatal flex scanner internal error--end of buffer missed
```

To reenter the scanner, first use

```
yyrestart( yyin );
```

Note that this call will throw away any buffered input; usually this isn't a problem with an interactive scanner.

Also note that `flex` C++ scanner classes are *reentrant*, so if using C++ is an option for you, you should use them instead. See "Generating C++ Scanners" above for details.

- `output()` is not supported. Output from the `ECHO` macro is done to the file-pointer `yyout` (the default being `stdout`).

  `output()` is not part of the POSIX specification.
- `lex` does not support exclusive start conditions (`%x`), though they are in the POSIX specification.
- When definitions are expanded, `flex` encloses them in parentheses. With `lex`, the following:

```
NAME    [A-Z][A-Z0-9]*
%%
foo{NAME}?      printf( "Found it\n" );
%%
```

will not match the string `foo` because when the macro is expanded the rule is equivalent to

```
foo[A-Z][A-Z0-9]*?
```

and the precedence is such that the '?' is associated with [A-Z0-9]* character sequence. With flex, the rule will be expanded to

```
foo([A-Z][A-Z0-9]*)?
```

and so the string foo will match.

Note that if the definition begins with ^ or ends with $, then it is *not* expanded with parentheses, to allow these operators to appear in definitions without losing their special meanings. But the <s>, /, and <<EOF>> operators cannot be used in a flex definition.

Using **-l** results in the lex behavior of no parentheses around the definition.

The POSIX specification is that the definition be enclosed in parentheses.

- Some implementations of lex allow a rule's action to begin on a separate line, if the rule's pattern has trailing whitespace:

```
%%
foo|bar<space here>
{ foobar_action(); }
```

  flex does not support this feature.

- The lex %r (generate a Ratfor scanner) option is not supported. It is not part of the POSIX specification.

- After a call to unput(), yytext is undefined until the next token is matched, unless the scanner was built using %array. This is not the case with lex or the POSIX specification. The **-l** option does away with this incompatibility.

- The precedence of the {} (numeric range) operator is different. lex interprets abc{1,3} as match one, two, or three occurrences of the character sequence 'abc', whereas flex interprets it as

```
match 'ab'
```

  followed by one, two, or three occurrences of 'c'. The latter is in agreement with the POSIX specification.

- The precedence of the ^ operator is different. lex interprets ^foo|bar as match either 'foo' at the beginning of a line, or 'bar' anywhere, whereas flex interprets it as match either 'foo' or 'bar' if they come at the beginning of a line. The latter is in agreement with the POSIX specification.

- The special table-size declarations such as %a supported by lex are not required by flex scanners; flex ignores them.
- The name FLEX_SCANNER is given a value with a #define directive so scanners may be written for use with either flex or lex. Scanners also include YY_FLEX_MAJOR_VERSION and YY_FLEX_MINOR_VERSION indicating which version of flex generated the scanner (for example, for the 2.5 release, these defines would be 2 and 5 respectively).

The following flex features are not included in lex or the POSIX specification:

C++ scanners (generation of)

%option

start condition scopes

start condition stacks

-I, -B (interactive/non-interactive scanners)

yy_scan_string()
(and friends)

yyterminate()

yy_set_interactive()

yy_set_bol()

YY_AT_BOL()

<<EOF>>

<*>

YY_DECL

YY_START

YY_USER_ACTION

`YY_USER_INIT`

`#line` (line directives)

`%{}` (any actions enclosed in)

multiple actions on a line

plus almost all of the `flex` flags. The last feature in the list refers to the fact that with `flex` you can put multiple actions on the same line, separated with semicolons, while with `lex`, the following

```
foo    handle_foo(); ++num_foos_seen;
```

is (rather surprisingly) truncated to

```
foo    handle_foo();
```

`flex` does not truncate the action. Actions that are not enclosed in braces are simply terminated at the end of the line.

DIAGNOSTICS

`warning, rule cannot be matched`

indicates that the given rule cannot be matched because it follows other rules that will always match the same text as it. For example, in the following example, `foo` cannot be matched because it comes after an identifier catch-all rule:

```
[a-z]+    got_identifier();
foo       got_foo();
```

Using `REJECT` in a scanner suppresses this warning.

`warning, -s option given but default rule can be matched`

means that it is possible (perhaps only in a particular start condition) that the default rule (match any single character) is the only one that will match a particular input. Since **-s** was given, presumably this is not intended.

`reject_used_but_not_detected undefined`
`yymore_used_but_not_detected undefined`
These errors can occur at compile time. They indicate that the scanner uses `REJECT` or `yymore()` but that `flex` failed to notice the fact, meaning that `flex` scanned the first two sections looking for occurrences of these actions and failed to find any, but somehow you snuck some in (via a `#include` file, for example). Use `%option reject` or `%option yymore` to indicate to `flex` that you really do use these features.

`flex scanner jammed`
a scanner compiled with **-s** has encountered an input string which wasn't matched by any of its rules. This error can also occur due to internal problems.

`token too large, exceeds YYLMAX`
your scanner uses `%array` and one of its rules matched a string longer than the `YYLMAX` constant (8K bytes by default). You can increase the value by using a `#define` directive to reset the value of `YYLMAX` in the definitions section of your `flex` input.

`scanner requires -8 flag to use the character` *'x'*
Your scanner specification includes recognizing the 8-bit character 'x' and you did not specify the **-8** flag, and your scanner defaulted to 7-bit because you used the **-Cf** or **-CF** table compression options. See the discussion of the **-7** flag for details.

`flex scanner push-back overflow`
you used `unput()` to push back so much text that the scanner's buffer could not hold both the pushed-back text and the current token in `yytext`. Ideally the scanner should dynamically resize the buffer in this case, but at present it does not.

`input buffer overflow, can't enlarge buffer because scanner uses REJECT`
the scanner was working on matching an extremely large token and needed to expand the input buffer. This doesn't work with scanners that use `REJECT`.

`fatal flex scanner internal error--end of buffer missed`
: This can occur in an scanner which is reentered after a long-jump has jumped out (or over) the scanner's activation frame. Before reentering the scanner, use:

```
yyrestart( yyin );
```

or, as noted above, switch to using the C++ scanner class.

`too many start conditions in <> construct!`
: you listed more start conditions in a **<>** construct than exist (so you must have listed at least one of them twice).

FILES

`-lfl`	library with which scanners must be linked.
`lex.yy.c`	generated scanner (called `lexyy.c` on some systems).
`lex.yy.cc`	generated C++ scanner class, when using **-+**.
`<FlexLexer.h>`	header file defining the C++ scanner base class, `FlexLexer`, and its derived class, `yyFlexLexer`.
`flex.skl`	skeleton scanner. This file is only used when building `flex`, not when `flex` executes.
`lex.backup`	backing-up information for **-b** flag (called `lex.bck` on some systems).

DEFICIENCIES / BUGS

Some trailing context patterns cannot be properly matched and generate warning messages (`dangerous trailing context`). These are patterns where the ending of the first part of the rule matches the beginning of the second part, such as

```
zx*/xy*
```

where the'`x*`' matches the '`x`' at the beginning of the trailing context. (Note that the POSIX draft states that the text matched by such patterns is undefined.)

For some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned performance loss. In particular, parts using | or `{n}` (such as `foo{3}`) are always considered variable-length.

Combining trailing context with the special '|' action can result in *fixed* trailing context being turned into the more expensive *variable* trailing context. For example, in the following:

```
%%
abc      |
xyz/def
```

Use of `unput()` invalidates `yytext` and `yyleng`, unless the `%array` directive or the **`-l`** option has been used.

Pattern-matching of NUL values is substantially slower than matching other characters.

Dynamic resizing of the input buffer is slow, as it entails rescanning all the text matched so far by the current (generally huge) token.

Due to both buffering of input and read-ahead, you cannot intermix calls to `<stdio.h>` routines, such as, for example, `getchar()`, with `flex` rules and expect it to work. Call `input()` instead.

The total table entries listed by the **-v** flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states if the scanner does not use `REJECT`, and somewhat greater than the number of states if it does.

`REJECT` cannot be used with the **-f** or **-F** options.

The `flex` internal algorithms need documentation.

SEE ALSO

Utility Programs -	`gawk`, `sed`, `byacc`
Documents -	*Lex & Yacc,* John Levine, Tony Mason, and Doug Brown,
	LEX - Lexical Analyzer Generator, M. E. Lesk and E. Schmidt
	Compilers: Principles, Techniques and Tools, Alfred Aho, Ravi Sethi and Jeffrey Ullman. Describes the pattern-matching techniques used by `flex` (deterministic finite automata).

AUTHOR

Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many `flex` beta-testers, feedbackers, and contributors, especially Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, benson@odi.com, Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko

Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, ken@ken.hilco.com, Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ support; to Kent Williams and Tom Epperly for C++ class support; to Ove Ewerlid for support of NUL values; and to Eric Hughes for support of multiple buffers.

This work was primarily done when I was with the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to `vern@ee.lbl.gov`.

NAME

`flook` – interactively examine file blocks

SYNOPSIS

`flook` *file*

DESCRIPTION

`flook` is a simple hexadecimal dump program which allows interactive examination of files. The user is prompted for block numbers, which should be given in hex. `flook` displays each 512-byte block as a matrix of 8 character long integers. The program is stopped by generating EOF (usually **`^D`**) on the keyboard in response to a block number request.

SEE ALSO

Utility Programs - `od`

NAME

`fmtflop` – format floppy diskettes

SYNOPSIS

`fmtflop [-f`*fill* **`-i`***interleave***`]`** *device*

DESCRIPTION

`fmtflop` formats media in floppy drives of various descriptions. All data previously on the diskette are lost.

OPTIONS

`-f` *fill* — The given *fill* value, between 0 and 255, is used to initialize blocks. The default fill value depends upon the underlying hardware; zero should be invariably avoided.

`-i` *interleave* — Sets the block *interleave* for the disk. The best possible value is one; for some devices, larger numbers may give better performance due to slow controller hardware.

`-v` — Generates verbose output.

`-q` — Prevents verification of the newly formatted disk.

SEE ALSO

Utility Programs - `mkfs`

System Calls - `ioctl()`

NAME

`fmtscsi` – format SCSI disk drives

SYNOPSIS

fmtscsi [-ceFv] [-i *interleave* **]** *device*

DESCRIPTION

`fmtscsi` performs various maintenance functions on SCSI disk devices, including formatting. In actuality, because many SCSI disks do nothing in response to a format request, `fmtscsi` is more often used to configure programmable options of the SCSI drive controller.

OPTIONS

-c The RAM disk cache (if any) on the SCSI drive is enabled. There is no way to disable the cache.

-e Configures the "error recovery parameters" of the controller. These are set to a value thought to be optimal for use under LynxOS.

-F Formats the disk. As mentioned above, this command is ignored by many devices.

-v Requests more output from other commands. Normally, **-c** and **-e** are silent unless they fail.

If **-v** is used with no other options, `fmtscsi` produces a report showing much information about the device. Included in the report are several categories of values.

Geometry Values:

Cylinders	Number of cylinders on the disk.
Heads	Number of heads (actually, the number of data surfaces).

Format Values:

`Trk/Z`	Tracks per zone. A "zone" is the unit of error management, and is always one or more tracks.
`A-Sec/Z`	Alternate sectors per zone. Each error recovery zone is allocated some number of alternate sectors in case bad sectors are detected.
`A-Trk/Z`	Alternate tracks per zone. Some drives allocate a complete spare track for each zone.
`A-Trk/V`	Alternate tracks per volume. Spare tracks are used as bad sector overflow areas in case some zones run out of spare sectors.
`Sec/Trk`	Sectors per track. Some SCSI drives actually support varying numbers of sectors depending on the radial position of the track. Such drives may report the average value of sectors per track, or may report something meaningless.
`Byte/Sec`	Bytes per sector. This value should be 512 bytes.
`I`	Interleave. Most SCSI drives have enough muscle in the controller to support an interleave of 1.
`T-skew`	Track skew. This value represents a sector number offset to account for track-to-track delay of switching the active head.
`C-skew`	Cylinder skew. This value represents a sector number offset to account for head movement delay.

Cache Values:

`Enabled`	Either "yes" or "no".
`Size`	Size of the cache in sectors.

`Prefetch Threshold`
: The maximum size (in sectors) of a "read" request which will trigger a prefetch to fill the cache.

`Max Prefetch`
: The maximum number of sectors which will be prefetched by the drive upon any read request. The value is displayed as `n+m*R`, meaning that the maximum value is *n* plus the number of blocks requested ("`R`") times *m*.

`Min Prefetch`
: The minimum number of sectors which will be prefetched by the drive upon a read request, expressed like the maximum value above.

Error Recovery Values:

`AWRE` Automatic Write Reallocation Enabled. If set to 1, the drive will relocate bad blocks detected during write operations.

`ARRE` Automatic Read Reallocation Enabled. If set to 1, the drive will relocate bad blocks detected during read operations.

`TB` Transfer Block. When set to 1, the drive will transfer a failing data block after a read request.

`RC` Read Continuous. When set to 1, the drive will transfer data even when errors are detected in order to maintain a continuous stream.

`EEC` Enable Early Correction. When set to 1, the drive will apply error correction to a sector as soon as possible, before the retry count is exhausted.

`PER` Post Error. When set to 1, the drive reports an error to the initiator after any errors are detected (even corrected errors).

`DTE`	Disable Transfer on Error. When set to 1 when `PER` is also 1, causes a transfer to be terminated when an error is detected, even if the error is correctable.
`DCR`	Disable Correction. When set to 1, all `ECC` corrections are disabled.

The current defect count is also shown. The meaning of this value varies from drive to drive, and is probably only worth noticing if it is large.

`-i` *interleave* Sets the block interleave for the disk. The best possible value is one; for some devices, larger numbers may give better performance due to slow controller hardware.

SEE ALSO

Utility Programs - `mkfs`

System Calls - `ioctl()`

NAME

`fsck` – file system check and repair

SYNOPSIS

`fsck [-amv]` *block_device*

DESCRIPTION

`fsck` checks and repairs the file system on the named *block_device*. It looks for inconsistencies among the following file system components:

Inode contents	All information in each active inode is checked for consistency. This includes the file type, the block pointers, the number of blocks in the file, and the file size in bytes. Cross-checks are made to ensure that each block is owned by at most one inode. Any inconsistencies are repaired if possible. Inodes beyond repair are zeroed and added to the free list.
Directories	Directory contents are checked for correctness and consistency. This includes the lengths of directory entries, the lengths of file names, and the existence of exactly one "." entry and exactly one ".." entry per directory. Reference counts for each inode are recomputed and repaired if necessary.

Any inconsistencies are reported and repaired. When `fsck` finishes checking for inconsistencies, it searches for files whose directory entry references have been lost. Such files are known as orphaned files. If `fsck` detects any orphaned files, it will create a `/lost+found` directory and add all orphaned files to `/lost+found` with file names set to their inode numbers. Orphaned directories are added intact to `/lost+found` with all of their contents, including subdirectories.

When `fsck` is done checking for orphaned files, it creates the bitmapped free block list and rebuilds the free inode list in order of inode number. If the file system is not FLFS (Fast Lynx File System), and the **`-m`** option is not specified, then `fsck` does not build a bitmapped free block list, and instead builds a free block list in order of block number.

OPTIONS

-a	Display error messages for any access, modification, and status times that are in the future.
-m	Create a bitmapped free block list on a non-FLFS file system.
-v	Verbose. Print each file/directory as it is checked.

NAME

`ftp` – file transfer program

SYNOPSIS

ftp [-v] [-d] [-i] [-n] [-g] [*host*]

DESCRIPTION

`ftp` is the user interface to the ARPANET standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which `ftp` is to communicate may be specified on the command line. If this is done, `ftp` will immediately attempt to establish a connection to an FTP server on that host; otherwise, `ftp` will enter its command interpreter and await instructions from the user.

COMMANDS

When `ftp` is awaiting commands from the user, the prompt "`ftp`>" is provided. The following commands are recognized by `ftp`:

! Invoke a shell on the local machine.

append *local-file* [*remote-file*]
Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for `type`, `format`, `mode`, and `structure`.

ascii Set the file transfer `type` to network ASCII. This is the default type.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer type to support binary image transfer.

bye Terminate the FTP session with the remote server and exit `ftp`.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

close Terminate the FTP session with the remote server, and return to the command interpreter.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified it is used to set the debugging level. When debugging is on, `ftp` prints each command sent to the remote machine, preceded by the string "**->**".

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, output comes to the terminal.

form *format* Set the file transfer `form` to *format*. The default format is "file".

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the `local-file` name is not specified, it is given the same name it has on the remote machine. The current settings for `type`, `form`, `mode`, and `structure` are used while transferring the file.

hash Toggle hash-sign ("**#**") printing for each data block transferred. The size of a data block is 1024 bytes.

glob Toggle file name globbing. With file name globbing enabled, each local file or pathname is processed for shell meta-characters. These characters include "*****", "**?**", "**[**", "**]**", " ", "**{**", "**}**". Remote files specified in multiple item commands (e.g., `mput`, are globbed by the remote server). With globbing disabled all files and pathnames are treated literally.

help [*command*]
: *Print* an informative message about the meaning of *command*. If no argument is given, `ftp` prints a list of the known commands.

lcd [*directory*]
: Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
: Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If no local file is specified, the output is sent to the terminal.

mdelete *remote-files*
: Delete the specified files on the remote machine. If globbing is enabled, the specification of remote files will first be expanded using `ls`.

mdir *remote-files local-file*
: Obtain a directory listing of multiple files on the remote machine and place the result in *local-file.*

mget *remote-files*
: Retrieve the specified files from the remote machine and place them in the current local directory. If globbing is enabled, the specification of remote files will first be expanding using `ls`.

mkdir *directory-name*
: Make a directory on the remote machine.

mls *remote-files local-file*
: Obtain an abbreviated listing of multiple files on the remote machine and place the result in local-file.

mode [*mode-name*]
: Set the file transfer `mode` to *mode-name*. The default mode is “stream” mode.

mput *local-files*
: Transfer multiple local files from the current local directory to the current working directory on the remote machine.

open *host* [*port*] Establish a connection to the FTP server *host*. An optional port number may be supplied, in which case, ftp will attempt to contact an FTP server at that port. If "auto-login" is enabled, ftp will also attempt to automatically log the user in to the FTP server (see below).

prompt Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default), any mget or mput will transfer all files.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If remote-file is left unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for type, format, mode, and structure.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for bye.

quote *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server. A single FTP reply code is expected in return.

recv *remote-file* [*local-file*]
A synonym for get.

remotehelp [*command-name*]
Request help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

rename [*from*] [*to*] Rename the file *from* on the remote machine, to the file *to*.

rmdir *directory-name*
Delete a directory on the remote machine.

send *local-file* [*remote-file*]
A synonym for put.

sendport Toggle the use of PORT commands. By default, `ftp` will attempt to use a PORT command when establishing a connection for each data transfer. If the PORT command fails, `ftp` will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

status Show the current status of `ftp`.

struct [*struct-name*]
Set the file transfer `structure` to *struct-name*. By default "stream" structure is used.

tenex Set the file transfer type to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer **type** to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, `ftp` will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. Unless `ftp` is invoked with "auto-login" disabled, this process is done automatically on initial connection to the FTP server.

verbose Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, verbose is on.

? [*command*] A synonym for help.

Command arguments which have embedded spaces may be quoted with double quote (") marks.

FILE NAMING CONVENTIONS

Files specified as arguments to `ftp` commands are processed according to the following rules.

1. If the file name "–" is specified, then `stdin` (for reading) or `stdout` (for writing) is used.
2. If the first character of the file name is "|", the remainder of the argument is interpreted as a shell command. `ftp` then forks a shell, using `popen(3)` with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; e.g., "`| ls -lt`". A particularly useful example of this mechanism is: "`dir | more`".
3. Failing the above checks, if "globbing" is enabled, local file names are expanded according to the rules used in the shell; `c.f.` the `glob` command.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The type may be one of "ascii", "image" (binary), "ebcdic", and "local byte size" (for PDP- 10's and PDP-20's mostly). `ftp` supports the ascii and image types of file transfer.

`ftp` supports only the default values for the remaining file transfer parameters: `mode`, `form`, and `struct`.

OPTIONS

Options may be specified at the command line, or to the command interpreter.

–v (Verbose on). Option forces `ftp` to show all responses from the remote server, as well as report on data transfer statistics.

-n	Option restrains `ftp` from attempting "auto-login" upon initial connection. If "auto-login" is enabled, `ftp` will check the `.netrc` file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, `ftp` will use the login name on the local machine as the user identity on the remote machine, and prompt for a password and, optionally, an account with which to login.
-i	Option turns off interactive prompting during multiple file transfers.
-d	Option enables debugging.
-g	Option disables file name globbing.

NOTE

Many FTP server implementation do not support the experimental operations such as print working directory.

Aborting a file transfer does not work right; if one attempts this the local `ftp` will likely have to be killed by hand.

ACKNOWLEDGEMENTS

This section was developed by the University of California, Berkeley.

FILES

`/etc/inetd.conf`

`$HOME/.netrc`

`/net/inetd`

SEE ALSO

Utility Programs - `ftpd, inetd`

File Formats - `hosts, inetd.conf, netrc`

NAME

`ftpd` – DARPA Internet File Transfer Protocol server

SYNOPSIS

`ftpd [-d ] [-l ] [-t` *timeout* **`] [-T`** *maxtimeout* **`]`**

DESCRIPTION

`ftpd` is the DARPA Internet File Transfer Protocol server process. The server uses the TCP protocol and listens at the port specified in the `ftp` service specification; see `services`.

If the **`-d`** option is specified, debugging information is written to the `syslog`.

If the **`-l`** option is specified, each `ftp` session is logged in the `syslog`.

The `ftp` server will timeout an inactive session after 15 minutes. If the **`-t`** option is specified, the inactivity timeout period will be set to *timeout* seconds. A client may also request a different timeout period; the maximum period allowed may be set to *timeout* seconds with the **`-T`** option. The default limit is 2 hours.

The `ftp` server currently supports the following `ftp` requests; case is not distinguished.

Request	Description
`ABOR`	Abort previous command
`ACCT`	Specify account (ignored)
`ALLO`	Allocate storage (vacuously)
`APPE`	Append to a file
`CDUP`	Change to parent of current working directory
`CWD`	Change working directory
`DELE`	Delete a file

Request	Description
`HELP`	Give help information
`LIST`	Give list files in a directory (`ls -lgA`)
`MKD`	Make a directory
`MDTM`	Show last modification time of file
`MODE`	Specify data transfer *mode*
`NLST`	Give name list of files in directory
`NOOP`	Do nothing
`PASS`	Specify password
`PASV`	Prepare for server-to-server transfer
`PORT`	Specify data connection port
`PWD`	Print the current working directory
`QUIT`	Terminate session
`RETR`	Retrieve a file
`RMD`	Remove a directory
`RNFR`	Specify rename-from file name
`RNTO`	Specify rename-to file name
`SITE`	Non-standard commands (see next section)
`SIZE`	Return size of file
`STAT`	Return status of server
`STOR`	Store a file
`STOU`	Store a file with a unique name
`STRU`	Specify data transfer `structure`
`SYST`	Show operating system type of server system
`TYPE`	Specify data transfer `type`
`USER`	Specify user name

Request	Description
XCUP	Change to parent of current working directory (deprecated)
XCWD	Change working directory (deprecated)
XMKD	Make a directory (deprecated)
XPWD	Print the current working directory (deprecated)
XRMD	Remove a directory (deprecated)

The following non-standard or UNIX specific commands are supported by the `SITE` request.

Request	Description
UMASK	Change `umask`, e.g. `SITE UMASK 002`
IDLE	Set idle-timer, e.g. `SITE IDLE 60`
CHMOD	Change mode of a file, e.g. `SITE CHMOD 755 filename`
HELP	Give help information, e.g. `SITE HELP`

The remaining `ftp` requests specified in Internet RFC 959 are recognized, but not implemented. `MDTM` and `SIZE` are not specified in RFC 959, but will appear in the next updated FTP RFC.

The `ftp` server will abort an active file transfer only when the `ABOR` command is preceded by a Telnet "Interrupt Process" (IP) signal and a Telnet "Synch" signal in the command Telnet stream, as described in Internet RFC 959. If a `STAT` command is received during a data transfer, preceded by a Telnet IP and Synch, transfer status will be returned.

`ftpd` interprets file names according to the "globbing" conventions used by `csh`. This allows users to utilize the meta-characters **" * ? [ ] { } .**

`ftpd` authenticates users according to three rules.

1. The user name must be in the password data base, `/etc/passwd`, and not have a null password. In this case a password must be provided by the client before any file operations may be performed.
2. The user name must not appear in the file `/etc/ftpusers`.
3. The user must have a standard shell returned by `getusershell`.
4. If the user name is “anonymous” or `ftp`, an anonymous `ftp` account must be present in the password file (user `ftp`). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host’s name).

In the last case, `ftpd` takes special measures to restrict the client's access privileges. The server performs a `chroot` command to the home directory of the `ftp` user. In order that system security is not breached, it is recommended that the `ftp` subtree be constructed with care; the following rules are recommended.

`~ftp)` Make the home directory owned by `ftp` and unwritable by anyone.

`~ftp/bin)` Make this directory owned by the super-user and unwritable by anyone. The program `ls` must be present to support the list command. This program should have mode `111`.

`~ftp/etc)]` Make this directory owned by the super-user and unwritable by anyone. The files `passwd` and `group` must be present for the `ls` command to be able to produce owner names rather than numbers. The password field in `passwd` is not used, and should not contain real encrypted passwords. These files should be mode `444`.

`~ftp/pub)]` Make this directory mode `777` and owned by `ftp`. Users should then place files which are to be accessible via the anonymous account in this directory.

NOTE

The anonymous account is inherently dangerous and should be avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user ID of the logged in user, reverting to the super-user only when binding addresses to sockets. The possible security holes have been extensively scrutinized, but are possibly incomplete.

ACKNOWLEDGEMENTS

This section was developed by the University of California, Berkeley.

FILES

`/etc/inetd.conf`

`/etc/hosts`

`/net/inetd`

SEE ALSO

Utility Programs - `ftp, getusershell, syslogd`

File Formats - `services, inetd.conf, hosts`